Mini-Curso: Usando Bancos de Dados Gratuitos com Delphi, MaxDB e SQLite

Clube Delphi

Delphi Prism
Data Binding no WPF

Boas Práticas
Utilizando diagramas UML no Delphi XE

Padrões de Projeto
Singleton na prática

Ano 9 :: Edição 133 :: R$ 11,90

DevMedia

ISSN 1517990-7

9 771517 990009 00133

Inversão de Controle

Injeção de Dependência

Dificuldades em encontrar conteúdo sobre .Net para iniciantes?

A DevMedia simplifica pra você!

easy .net magazine

Easy .Net Magazine:

A primeira revista digital feita especialmente para quem está dando os primeiros passos em .Net!

Você encontrará exemplos e modelos fáceis e práticos, noções introdutórias e conceitos básicos.

Conheça as vantagens:

- Vídeos e apresentações "embutidos" nas páginas;
- Ilustrações, diagramas e modelos apresentados em PowerPoint;
- Entrevistas em vídeo;
- Códigos-fonte que podem ser copiados com CTRL+C/CTRL+V.

A Easy.Net não é feita em PDF ou Flash!
Ela utiliza um aplicativo 100% online.

DevMedia

Acesse agora a edição Nº 1 totalmente grátis e comprove

www.devmedia.com.br/easynet

Easy.Net Magazine, seu curso inicial de .Net!

Sumário

Delphi

Delphi — **06 – Data Binding no WPF com Delphi Prism**
[*Fabrício Hissao Kawata*]

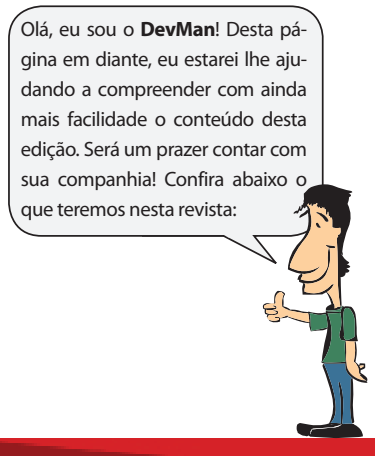
Mini-Curso — **13 – Bancos de Dados Gratuitos – Parte 3**
[*Vitor Luiz Rubio*]

Boas Práticas

Boas Práticas — **27 – Utilizando diagramas UML no Delphi XE**
[*Fabrício Hissao Kawata*]

Boas Práticas — **35 – Inversão de Controle e Injeção de Dependência**
[*Rafael Stavarengo*]

Boas Práticas — **43 – Usando o Padrão Singleton**
[*Vitor Luiz Rubio*]



Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Conhecimento faz diferença!

Faça já sua assinatura digital ! | www.devmedia.com.br/es

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre ouutros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DevMedia

Clube Delphi

Ano 12 - 133ª Edição 2011 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral
Guinther Pauli
guinther@devmedia.com.br

Editor Técnico
Paulo Quicoli
pauloquicoli@devmedia.com.br

Leandro Daniel
contato@leandrodaniel.com

Equipe Editorial
Vitor Luiz Rubio
vitorrubio@gmail.com

Rafael Stavarengo
faelsta@gmail.com

Jornalista Responsável
Kaline Dolabella - JP24185

Atendimento ao Leitor
A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/mancad
(21) 3382-5038

Publicidade
Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição
FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther@devmedia.com.br

EDITORIAL

Conceber aplicações de software bem estruturadas e flexíveis a mudanças tem sido um dos grandes desafios impostos à área de desenvolvimento de sistemas no decorrer dos anos. As organizações estão cada vez mais sujeitas a transformações repentinas e de profundo impacto. Isto acaba por exigir dos profissionais de Tecnologia da Informação uma resposta rápida ao atendimento de novas demandas ou, ainda, à adaptação de soluções pré-existentes.

Prazos exíguos, equipes reduzidas, falta de planejamento, inexistência de uma modelagem para a solução a ser entregue, dentre outros aspectos, quase sempre resultarão em codificação de baixa qualidade para a solução que será obtida. Isto pode se constituir, futuramente, num verdadeiro pesadelo em situações que exijam adaptações de funcionalidades já criadas anteriormente.

A utilização da técnica de desenvolvimento em camadas, prática amplamente difundida no mercado, procura fornecer meios para a criação de aplicações mais flexíveis. Entretanto, a simples utilização deste tipo de abordagem não garante por si só resultados positivos. Ainda existirá a possibilidade de se chegar a um produto que faz uso de classes de objetos com um alto grau de acoplamento; com isto, ajustes que na teoria seriam simples levam a mudanças drásticas em todo um software, aumentando ainda a chance de se introduzirem erros inesperados na solução. A Injeção de Dependência auxilia a resolver esse dilema, desacoplando as camadas criadas.

Primeiramente, deve-se levar em conta que a Injeção de Dependência (DI, do inglês Dependency Injection) é na verdade, uma aplicação específica de um conceito mais abrangente conhecido como Inversão de Controle (IoC, do inglês Inversion of Control).

A Inversão de Controle é uma técnica na qual um objeto delega a execução de uma atividade a outra classe, ao invés de tentar controlar diversos aspectos que fugiriam do objetivo inicial para o qual havia sido definido. Esta característica é particularmente importante, já que permite o desenvolvimento de classes com um maior grau de coesão, visto que não acumulam um grande número de responsabilidades, o que do contrário dificultaria a manutenção futura das mesmas e resultaria em um software mais propenso a falhas.

O uso de injeção de dependência também é uma forma de se aplicar o conceito de inversão de controle. Neste caso ao invés de um objeto instanciar todas as classes das quais depende, o mesmo delega a outro mecanismo esta tarefa (container). Ao se optar pela utilização de um container o mesmo será responsável pelas seguintes atividades: Analisar de quais tipos de objetos uma classe depende; Criar os objetos a serem consumidos; Associar as instâncias de objetos criadas à classe consumidora, sendo que a isto se chama "resolver" as dependências.

Dentre as vantagens da adoção de um framework de injeção de dependência, merece ser destacado: Foco no desenvolvimento voltado ao uso de interfaces, com a obtenção de aplicações em que os componentes tendem a ser mais coesos, além de estarem fracamente acoplados entre si, já que o conceito de Injeção de Dependência privilegia o desenvolvimento de software em camadas; Flexibilidade diante da necessidade de mudanças, devido à redução na dependência entre as partes que constituem uma aplicação; O desenvolvimento de software seguindo boas práticas, uma vez que os conceitos de Inversão de Controle e Injeção de Dependência promovem o uso de patterns amplamente difundidos e utilizados com sucesso pela comunidade de desenvolvedores; Evita-se a duplicação de código instanciando classes e que estaria espalhado por toda uma solução.

Inversão de Controle e Injeção de Dependência são os temas de destaque desta edição da ClubeDelphi, que você vai poder conferir no artigo do Rafael Stavarengo.

Guinther Pauli
guinther@devmedia.com.br
Certificado Delphi 3,5,6,7,2005,2006,Web,Linux
Microsoft Certified: MCP, MCAD, MCSD.NET, MCTS, MCPD (ASP.NET,C#)
guintherpauli.blogspot.com - twitter.com/guintherpauli

ClubeDelphi PLUS

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

DevMedia

Sistema de créditos DevMedia

Conteúdo **completo** ao seu alcance

Com o sistema de créditos DevMedia você pode comprar individualmente mais de 4.000 vídeo aulas, 8000 artigos, além de palestras e vídeos com formações completas.

TODAS as vídeo aulas que você sempre quis ver a partir de

R$ 0,99

ECONOMIA: você não precisa assinar 8 produtos diferentes para ter acesso ao conteúdo completo do site!

Saiba mais sobre o Sistema de Créditos!

www.devmedia.com.br/creditos

DevMedia

Data Binding no WPF

No cenário atual, o WPF se estabeleceu como a principal opção para desenvolvimento Desktop na plataforma .NET, com o foco voltado para aplicações de interface rica. Com o uso do Delphi Prism, ganha-se mais esta opção no desenvolvimento Win32, em vista das já consagradas aplicações VCL Forms, tradicionais do Delphi. Dessa forma, o uso de Data Binding torna-se imprescindível para que se ganhe a agilidade necessária para, por exemplo, a vinculação dinâmica entre os atributos de um objeto de negócio aos elementos da interface.

Dentre seus inúmeros recursos, o Windows Presentation Foundation apresenta uma forma simplificada, mas não menos consistente no que diz respeito à interação e apresentação de dados em aplicações. Isso quer dizer que seus elementos podem estar ligados a fontes de dados distintas, mais precisamente a qualquer objeto .NET. Como exemplo, podemos citar um simples arquivo XML, uma tabela de um banco de dados, um serviço web, um objeto de negócio ou ainda um próprio elemento WPF. Este processo é denominado Data Binding, que numa tradução livre pode ser interpretado como ligação de dados.

Na prática, este termo representa basicamente a conexão estabelecida entre a interface visual (UI) da aplicação e a lógica de negócios. Para que esta associação se dê de maneira ideal, são necessárias algumas configurações estipuladas pelo WPF de forma a tornar o processo de notificação adequado, para que quando os dados da fonte sofram algum tipo de mudança, os elementos vinculados a ele consigam refletir estas alterações automaticamente. Num caso oposto, as informações editadas num elemento da UI, devem ser refletidas na fonte.

Em suma, no WPF o Data Binding permite definir, através de uma simples declaração, como os dados pertinentes à aplicação irão interagir com os diversos elementos e controles presentes em sua interface. Nesta simples definição, o termo "declaração" é usado em vista das duas formas possíveis de se obter uma ligação de dados: declarativa e imperativa. Na abordagem imperativa, se faz necessário o uso de comando(s) que deverão ser executados para que se estabeleça um vínculo de dados, enquanto que na forma declarativa é apenas descrita a relação que a fonte de dados terá com seu destino (como, por exemplo, um elemento da UI).

No campo prático, imagine o cenário onde seja necessário mostrar os itens de uma coleção em um controle adequado, com um ListBox. Utilizando a forma imperativa, teríamos código para diversas etapas do processo, como para a recuperação do conjunto de dados necessários (que são os próprios itens da coleção), para a iteração destes itens a por fim outro código para o preenchimento do controle ListBox. Ao mesmo tempo, utilizando o recurso de Data Binding do WPF, uma simples declaração de código, seja procedural (Oxygene) ou de marcação (XAML) bastaria para se estabelecer toda a relação entre coleção e controle.

Resumo DevMan

De que trata o artigo:

Este artigo está fundamentado em apresentar, de forma geral, o recurso de Data Binding do WPF (Windows Presentation Foundation), que lida com a ligação de dados entre extremos, determinados entre fonte e destino. Um exemplo clássico é a representação dos itens de uma coleção de dados em um controle visual, estabelecendo um vínculo entre eles de tal forma a manter os dados sincronizados em tela, conforme as mudanças ocorrem na coleção.

Em que situação o tema é útil:

Seja qual for o segmento da aplicação que está se desenvolvendo com WPF, em dado momento será necessário o uso do recurso de Data Binding. Em uma situação do mundo real, tomando como base a necessidade de se apresentar visualmente em um controle ListBox o conjunto de dados contidos em um arquivo XML, a utilização de Data Binding torna este processo totalmente dinâmico. Em outras palavras, não se faz necessária uma intervenção manual, por conta da iteração dos itens da fonte (arquivo XML) e a adição um a um ao destino (ListBox).

Data Binding no WPF:

O maçante provimento de informações (artigos, vídeos, blogs, sites e textos avulsos) sobre o desenvolvimento com Delphi Prism fez com que a comunidade voltasse suas atenções também para este novo foco. Sendo assim, diante do mar de novas possibilidades proporcionadas pelo .NET Framework, o desenvolvimento de aplicações utilizando o Windows Presentation Foundation (WPF) se torna uma opção viável, surgindo como uma alternativa às já consagradas aplicações VCL Forms do Delphi tradicional. Diante disso, é fácil presumir que a nova forma de desenvolvimento (como agilidade e recursos disponíveis) muito provavelmente será comparada à forma tradicional à qual se está acostumada a trabalhar. Assim sendo, vários quesitos serão desejados, dentre eles a possibilidade de ligação de dados. Enquanto no Delphi podemos ter campos de tabelas de banco de dados vinculados a controles visuais (DBEdit, DBGrid etc.), de forma semelhante podemos ter no WPF os atributos das classes de negócio ligados aos elementos da interface, utilizando Data Binding, como será visto no decorrer do artigo.

com Delphi Prism

Fabrício Hissao Kawata

Data Binding

O Windows Presentation Foundation tem como uma de suas principais características o uso de uma linguagem de marcação denominada XAML (eXtensible Application Markup Language). Através dela, dá-se a forma mais comum de se declarar um Binding em aplicações WPF. Diz-se mais comum pelo fato de também ser possível a declaração via código procedural, utilizando a linguagem Oxygene, no caso do Delphi Prism.

Mas voltando ao XAML, para se declarar um Binding é necessário a utilização de um recurso chamado Markup Extensions, ou extensões de marcação. Markup Extensions são utilizadas para especificar valores, através de marcação, que fogem do âmbito padrão do XAML. No momento da compilação estes valores são então interpretados e analisados pela aplicação. Dessa forma, são identificados de maneira especial no corpo do código utilizando para isso um par de chaves ({...}) em sua declaração.

Ainda sobre Data Binding, além do uso de extensões de marcação, outra característica é a presença da palavra Binding em sua definição, conforme visto a seguir. Neste trecho é declarado um Binding da propriedade Content de um elemento Label com a propriedade Text de outro elemento TextBox:

```
<Label Name="Label1"
    Content="{Binding ElementName=TextBox1, Path=Text}"
    Height="28" Width="137" Margin="129,137,0,0"
    HorizontalAlignment="Left" VerticalAlignment="Top" />
```

De acordo com a sintaxe base utilizada para a ligação de dados, fora a denominação Binding, foram necessárias a declaração de outras duas propriedades: ElementName e Path. A primeira, ElementName, diz ao Binding qual elemento deve ser procurado no corpo do código XAML para ser usado como fonte de dados. Neste caso, é o elemento TextBox1. Já a segunda, Path, especifica o local exato de onde extrair as informações na fonte definida que, nesta declaração é a propriedade Text do elemento TextBox1.

Binding entre controles

No WPF, o ponto crucial para a ligação de dados (Data Binding) se dá pela classe System.Windows.Data.Binding. Um objeto deste tipo estabelece o vínculo entre duas propriedades, mantendo-as estritamente juntas, como se estivessem coladas. Dessa forma, mantém-se um canal de comunicação aberto entre ambas, podendo assim ser definido posteriormente a forma de sincronização que irá ser empregada durante o ciclo de vida da aplicação.

Para uma primeira demonstração prática, tomemos a necessidade de se mostrar o texto de um item selecionado em uma lista em outro controle. No caso, serão utilizados os controles TreeView e Label, conforme mostra a **Figura 1**. Inicialmente, a atividade será feita sem o uso real de Data Binding, ou seja, será utilizado um processo todo manual e imperativo, conforme é feito na **Listagem 1**.

Para que consigamos o resultado desejado, é codificado o evento SelectedItemChanged do controle TreeView, que é disparado sempre que um item de sua lista é selecionado. Neste exato momento, é atribuído ao conteúdo do controle Label o nome da revista marcada.

Listagem 1. Codificação do evento SelectedItemChanged do controle TreeView

```
method Window1.treeViewRevistas_SelectedItemChanged
  (sender: System. Object; e: System.Windows.RoutedPropertyChangedEvent
  Args<System.Object>);
begin
  labelNomeRevista.Content := (treeViewRevistas.SelectedItem as TreeView
  Item).Header.ToString();
end;
```

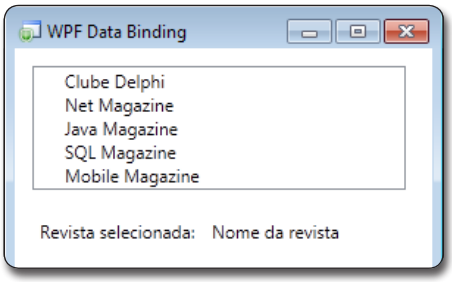


Figura 1. Aplicação sem uso de Data Binding

Com base no tema do artigo, pode-se presumir que todo este processo manual pode ser substituído por um processo específico e dinâmico, que é o de Data Binding. Portanto, em vista do que foi apresentado até aqui, duas mudanças serão fundamentais: será feito uso de um objeto Binding e toda a codificação será feita no contexto da janela (Window) e não mais no de um controle em específico. A **Listagem 2** mostra a codificação necessária que é feita no construtor da janela principal da aplicação.

Listagem 2. Utilizando Binding em nível de código

```
01 constructor Window1;
02 begin
03   InitializeComponent();
04
05   var binding1 := new Binding();
06   binding1.Source := treeViewRevistas;
07   binding1.Path := new PropertyPath("SelectedItem.Header");
08   labelNomeRevista.SetBinding(Label.ContentProperty, binding1);
09 end;
```

Conforme mencionado, neste código usa-se uma instância de um objeto Binding (System.Windows.Data.Binding) na linha 5. A seguir, são atribuídos valores para suas duas principais propriedades, no que diz respeito à ideal ligação de dados: Source e Path. A primeira refere-se ao objeto que será utilizado como fonte que, neste caso, trata-se do controle TreeView. Já o segundo, informa o caminho para a propriedade de origem que será utilizada na ligação, que é a propriedade Header do item selecionado do TreeView. Uma vez definido o objeto Binding, podemos atribuí-lo a outro objeto, de forma a estabelecer um vínculo entre ambos, como é feito com o controle Label, na linha 8.

Note que para isso é chamado o método SetBinding que é herdado por todos os elementos que descendem das classes FrameworkElement ou FrameworkContentElement. Ele tem por função anexar um objeto de Binding (binding1) a uma propriedade de dependência (Label.ContentProperty). Em suma, definimos assim a estrutura essencial para o uso de Data Binding que, como já citado anteriormente, envolve uma propriedade de origem e uma de destino.

Por questão de comparação, e já com o aplicativo em execução, ambas as abordagens apresentam essencialmente o mesmo resultado. No entanto, com um pouco mais de atenção notamos uma diferença sutil no momento da primeira execução da aplicação, no que diz respeito à exibição do texto do item selecionado no conteúdo do controle Label. Ao utilizar Data Binding, este conteúdo vem em branco, enquanto que na abordagem manual, este mesmo conteúdo apresenta o valor padrão definido no código XAML com os dizeres "Nome da revista".

Isto pode ser explicado pelo fato de que ao fazer uso de um objeto Binding, esta operação foi definida no construtor da janela da aplicação, fazendo com que o controle Label "labelNomeRevista" já inicie vinculado a ele. Como, por padrão, o TreeView é iniciado sem seleção alguma, o processo de binding falha, retornando um valor genérico que, neste caso, é uma string vazia. Já no outro processo o vínculo de dados não existe até que se tenha uma intervenção manual (seleção de um item), portanto de início o conteúdo do Label é mostrado conforme definido em seu XAML. A ligação, nesta situação, se dará no momento da seleção de um item do TreeView.

Outro ponto que pode ser notado neste momento é com relação à quantidade de código utilizado para se obter um mesmo resultado. Ao utilizarmos Data Binding, tivemos mais linhas sendo digitadas, o que pode parecer um fator negativo, num primeiro momento. No entanto, vale lembrar que o exemplo utilizado é minimamente complexo e que em situações de maiores proporções a agilidade que se ganha com o uso de Data Binding é enorme. Para citar algumas das vantagens que se pode obter, temos a própria economia de código e a centralização da definição de um objeto Binding que poderá atender a N objetos.

Nota do DevMan

FrameworkElement e FrameworkContentElement são as classes base para muitas classes importantes no Windows Presentation Foundation. Elas adicionam suporte a vertentes comuns entre os elementos WPF como, por exemplo, suporte a estilos (Styles), ligação de dados (Data Binding), mecanismos de dicas (Tooltips), menus de contexto(context menus), e outros recursos relativos a conteúdo (como imagens e texto).

Com base no que foi dito, o processo de Binding apresentado só poderá ser realizado por objetos derivados de FrameworkElement ou FrameworkContentElement. Em vista desta "restrição", uma outra alternativa de definição de Binding, é fazer uso de uma classe especial denominada BindingOperations que possui um método estático SetBinding. Este método possui a seguinte assinatura:

```
BindingOperations.SetBinding(target: DependencyObject, dp: DependencyProperty, binding: BindingBase)
```

Note que este método recebe como parâmetros, além dos objetos passados anteriormente, um parâmetro adicional na primeira posição que se refere ao objeto de destino. Graças a este parâmetro, é possível fazer o Binding também de objetos que não derivam das classes FrameworkElement e FrameworkContentElement. Portanto, com base no código apresentado na **Listagem 2**, sua linha 8 poderia ser substituída por:

```
BindingOperations.SetBinding(labelNomeRevista, Label.ContentProperty, binding1);
```

Removendo o Binding

Como vimos anteriormente, ao se definir um Binding a um objeto, este vínculo estabelecido permanece durante o ciclo de vida da aplicação. No entanto, pode haver casos especiais em que este vínculo necessite ser desfeito em determinado momento do ciclo. Em vista disso, ainda sobre o exemplo apresentado, ao se atribuir no construtor da janela, imediatamente após a definição do objeto Binding, um novo valor para a propriedade Content do controle labelNomeRevista, temos a perda do Data Binding. Logo, apesar de manual, esta é uma das formas de se remover o Binding de um objeto.

Obviamente, o WPF apresenta também alguns métodos próprios para isso, novamente através da classe BindingOperations. Esta, por sua vez, disponibiliza dois métodos estáticos com nomes bem sugestivos: ClearBinding e ClearAllBindings. Apesar de ambos terem a mesma função, a diferença entre um e outro é que o segundo é voltado para os objetos que possuem vínculo com vários Bindings, eliminando todas as ligações de uma só vez.

Sendo assim, a qualquer momento poderíamos fazer a chamada destes métodos para que o Binding em nosso exemplo fosse des-

feito. Facilmente, poderíamos direcioná-las ao evento Click de um botão, conforme mostra a **Listagem 3**. Veja que foram apresentadas as duas soluções, usando cada método citado (ClearBinding e ClearAllBindings), sendo que no primeiro caso se faz necessário especificar a propriedade de dependência (DependencyProperty) à qual se deseja remover o Binding.

Binding via XAML

Vimos até aqui o uso de Data Binding em nível de código procedural, que se deu no code-behind da janela XAML da aplicação, todavia, é possível também fazer esta definição no próprio código XAML. Neste novo plano, não temos mais o método SetBinding, e sim uma extensão de marcação (markup extension) para a declaração de Binding.

Vendo isto na prática, a ideia mais sensata é estabelecer uma comparação com o primeiro processo, definido via código. Com base nisso, o foco é conseguir o mesmo resultado anterior, agora com a definição do Binding no plano XAML. A **Listagem 4** fornece o trecho de código necessário para essa solução.

Listagem 3. Removendo o Binding de um objeto

```
method Window1.buttonRemoverBinding_Click(sender: System.Object;
e: System.Windows.RoutedEventArgs);
begin
 BindingOperations.ClearBinding(labelNomeRevista, Label.ContentProperty);
end;

method Window1.buttonRemoverAllBinding_Click(sender: System.Object;
e: System.Windows.RoutedEventArgs);
begin
 BindingOperations.ClearAllBindings(labelNomeRevista);
end;
```

Listagem 4. Definindo Binding em XAML

```
<Label Name="labelNomeRevista"
  Content="{Binding Source={x:Reference treeViewRevistas}, Path=
  SelectedItem.Header}"
.... />
```

Com esta nova codificação, fica nítido que em XAML, define-se o Binding diretamente na propriedade de destino do objeto alvo. No nosso caso, representado pela propriedade Content do objeto Label (labelNomeRevista). Outro adentro é com relação à referência para a fonte (Source) do objeto Binding, que agora é marcada como ElementName.

Neste ponto vale ressaltar que ElementName só deve ser usado para referenciar objetos descendentes de FrameworkElement e FrameworkContentElement. Para os outros casos, o WPF 4 introduz uma nova markup extension denominada "x:Reference" para referência de objetos. Dessa forma, podemos aproximar ainda mais o código XAML empregado ao código procedural usado, voltando a utilizar o atributo Source da seguinte forma:

```
<Label Name="labelNomeRevista"
    Content="{Binding Source={x:Reference treeViewRevistas},
    Path=SelectedItem.Header}"
.... />
```

No geral, empregar este tipo de codificação em XAML se apresenta mais simples e intuitivo, se comparado à forma procedural, no entanto o WPF possibilita as duas formas, deixando a cargo do desenvolvedor a escolha ideal.

No que tange à execução da aplicação, o mecanismo de Binding reflete numa propriedade destino exatamente o valor da propriedade de origem. Neste grupo de valores há de considerar também o valor null, ou seja, quando o valor provindo da origem é nulo. Para este tipo de situação, o WPF oferece uma propriedade específica que permite atribuir um valor padrão em substituição ao valor nulo. Esta propriedade é nomeada como TargetNullValue e seu emprego pode ser visto na **Listagem 5**. Para esta demonstração, é exposto um novo contexto onde além do valor da propriedade Header do item selecionado no TreeView, é feito o Binding de sua propriedade Tag. Com isso, para que o trabalho com TargetNullValue possa ser conferido, são atribuídos valores somente a alguns itens da coleção, semelhante ao trecho XAML da **Listagem 6**.

Listagem 5. Definindo a propriedade TargetNullValue

```
<Label Name="labelValorTag"
    Content="{Binding ElementName=treeViewRevistas,
              Path=SelectedItem.Tag,
              TargetNullValue=Tag com valor nulo}"
   .....
/>
```

Listagem 6. XAML com coleção

```
<TreeView Name="treeViewRevistas">
  <TreeViewItem Header="Clube Delphi" />
  <TreeViewItem Header="Net Magazine" Tag="1" />
  <TreeViewItem Header="Java Magazine" Tag="2" />
  <TreeViewItem Header="SQL Magazine" />
  <TreeViewItem Header="Mobile Magazine" Tag="3" />
</TreeView>
```

O resultado final pode ser visto na **Figura 2**, onde o item Clube Delphi aparece selecionado e, conforme definido anteriormente, não foi atribuído valor algum para a sua propriedade Tag. Num comportamento padrão, o Label que exibe o valor da Tag ficaria em branco, causando uma desconformidade visual na interface do aplicativo, bem como, em outros casos, vir a prejudicar uma regra de negócio que não permita valores nulos. Com a definição da propriedade de Binding TargetNullValue este fator é eficazmente tratado, mostrando um valor padrão definido em sua estrutura, que foi "Tag com valor nulo".

Data Binding - Two-Way

Nas amostragens abordadas até agora a forma de Binding utilizada se caracterizou pelo movimento dos dados de forma unidirecional, ou seja, a partir da leitura de uma fonte, um destino é atualizado. No entanto, há situações em que esta condição de vínculo se dê de forma bidirecional. Isto significa termos valores atualizados nas duas extremidades (fonte e destino) mediante a atualização em qualquer uma delas. No WPF, este processo é

conhecido como modo Two-Way do Data Binding, em que alterações realizadas na fonte atualizam o alvo, da mesma forma que mudanças no alvo são refletidas na fonte.

Para um melhor entendimento, suponhamos a seguinte situação: um aplicativo possui um medidor de volume sonoro e este é representado visualmente na aplicação de duas formas. A primeira, mais intuitiva, é caracterizada por um controle Slider, que permite através do movimento de deslize de seu indicador a mudança de valor do volume. A segunda forma se dá através de um controle TextBox comum, onde o valor do volume sonoro deve ser digitado. Em vista disso, o que se deseja é que o valor digitado na caixa de entrada seja refletido na posição do indicador do controle Slider, ao mesmo tempo em que, ao se deslizar o indicador deste, o valor marcado seja também devidamente refletido no controle TextBox. O visual desta aplicação de exemplo é representado pela **Figura 3**, onde são mostrados como os controles estão dispostos, para um melhor entendimento.

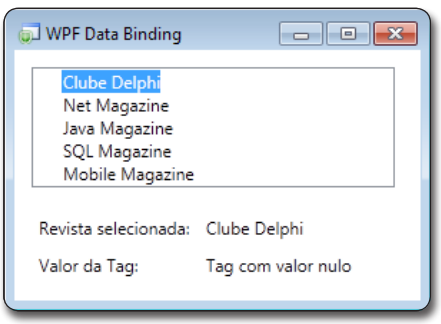


Figura 2. Utilizando a propriedade TargetNullValue

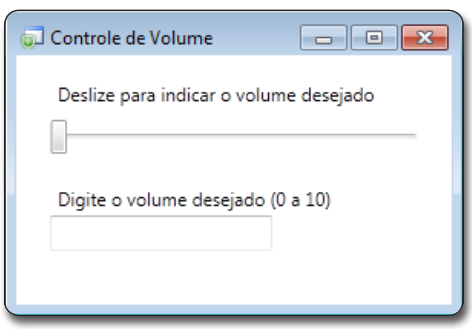


Figura 3. Aplicação de exemplo para Binding Two-Way

A seguir, no código da **Listagem 7** é apresentada a declaração do código XAML necessário para os controles Slider e TextBox. O diferencial, neste caso, se dá na linha 6, onde há a definição de Binding para a propriedade Text do controle TextBox. Na declaração de markup extension, além das propriedades vistas até este momento do artigo tem-se uma adicional denominada Mode. É nela que é indicado o modo com que se dará o fluxo dos dados entre fonte e alvo, definido aqui como Two-Way. Com esta aplicação em execução, o efeito desejado é obtido graças a apenas a intervenção em apenas uma única linha de código. A **Figura 4** demonstra um esboço deste resultado.

Não bastasse o ganho de produção apresentado neste exemplo, esse tipo de Binding ainda provê à aplicação alguns comportamentos inteligentes, tão necessários para o bom desempenho do software. Por exemplo, foi definido na marcação XAML que o intervalo de volume para o Slider vai de 0 a 10, através de seu valor mínimo e máximo.

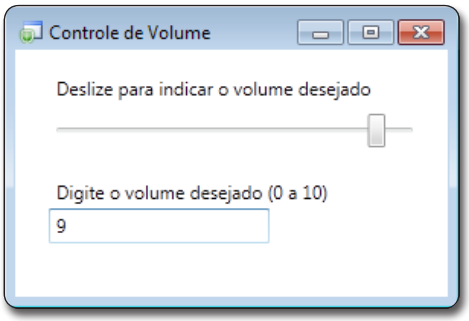


Figura 4. Aplicação de exemplo Two-Way em ação

Listagem 7. Declarando Binding no modo Two-Way

```
01  <Slider Name="sliderVolume"
02      Minimum="0" Maximum="10"
03      Height="23" Width="244" Margin="22,42,0,0"
04      HorizontalAlignment="Left" VerticalAlignment="Top" />
05  <TextBox Name="textBoxVolume"
06       Text="{Binding ElementName=sliderVolume, Path=Value,
         Mode=TwoWay}"
07       Height="23" Width="145" Margin="22,104,0,0"
08       HorizontalAlignment="Left" VerticalAlignment="Top" />
```

Ao mesmo tempo, em uma situação comum, o usuário poderia facilmente digitar um valor na caixa de entrada fora do limite permitido como, por exemplo, o número 15. Todavia, uma vez definido o Binding no modo Two-Way, o tratamento desta situação já acontece. Na prática, ao se informar no TextBox um valor maior que 10, o indicador do Slider tenta se posicionar no valor indicado, mas esbarra em seu limite máximo, que é 10. Imediatamente ao se posicionar neste novo ponto, o Binding atualiza o valor no TextBox, deixando-o com um valor dentro do aceitável, prevenindo a aplicação de erros.

Além do modo Two-Way, há ainda no WPF outros modos suportados pelo Data Binding:

- OneWay: neste modo, as alterações dos dados efetuadas na fonte (Source) serão refletidas no alvo;
- OneTime: o valor no alvo é atribuído apenas uma única vez, comumente quando o aplicativo é iniciado;
- OneWayToSource: neste modo, as alterações dos dados efetuadas no alvo são refletidas na fonte.

Ainda sobre o Mode do Data Binding, é importante ressaltar que, apesar de não ser obrigatório, é uma boa prática fazer a declaração explícita desta propriedade. Tão importante que a grande maioria dos livros sobre o assunto não deixam de mencionar esta observação. Isto se deve a dois fatores fundamentais. O primeiro e mais óbvio é sobre você ter o controle íntegro de sua aplicação de forma explícita, diminuindo a probabilidade de erros e comportamentos indesejados proporcionados pelo mecanismo de Binding.

Já o segundo fator justifica e complementa o primeiro, pois no WPF cada elemento possui um valor padrão para a propriedade Mode do Data Binding.

Só para citar, enquanto o padrão em um TextBlock é OneWay, em um TextBox é TwoWay. Em outras palavras, se omitirmos a declaração da propriedade Mode neste último exemplo, ainda assim teríamos o mesmo resultado. Já se num futuro este controle fosse substituído por outro, poderíamos ter problemas no comportamento ideal da aplicação e que seria facilmente solucionado pela declaração explícita da propriedade Mode.

Data Context

Como já dito, os controles WPF descendem em sua grande maioria da classe FrameworkElement. Esta classe, por sinal, provê a seus descendentes uma propriedade especial denominada Data Context, que está intimamente ligada ao assunto de Data Binding. Entenda Data Context, como o ponto de partida, ou seja, o contexto da ligação de dados. Tanto que, em vista do que foi apresentado até este momento, o conceito de Data Context já foi utilizado, embora não com esta nomenclatura. Na declaração de markup extension, ao utilizarmos a propriedade ElementName, estamos fornecendo um contexto para o Binding em questão. Ao mesmo tempo em que, através da propriedade Path, estamos especificando uma localização dentro do contexto.

Feita a analogia, através da declaração explícita de um Data Context, podemos ir além e definir, por exemplo, um mesmo contexto para toda uma hierarquia. Isso quer dizer que ao se definir a propriedade Data Context de um elemento, todos os seus filhos poderão utilizar este mesmo contexto.

A forma mais comum para se apresentar seu uso é através de um exemplo prático. Para este demonstrativo será utilizado um controle ListBox para listar verticalmente cada letra de uma palavra digitada numa caixa de entrada (TextBox). Um visual sugerido para esta aplicação de exemplo é mostrado na **Figura 5**.

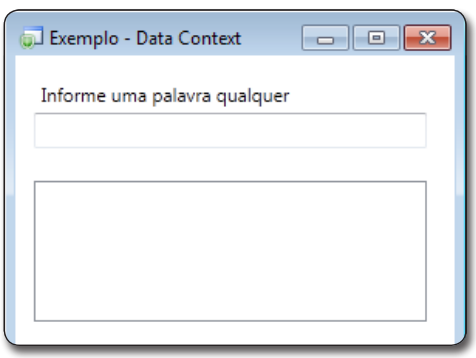


Figura 5. Aplicação de exemplo para Data Context

A função da aplicação será mostrar em tempo real no ListBox, através do uso de Data Binding, as letras que estão sendo usadas para formar a palavra que está sendo digitada naquele momento pelo usuário. Além disso, será apresentado como hint do controle ListBox o número total de letras utilizadas. Note que é necessário buscar duas informações diferentes numa mesma fonte.

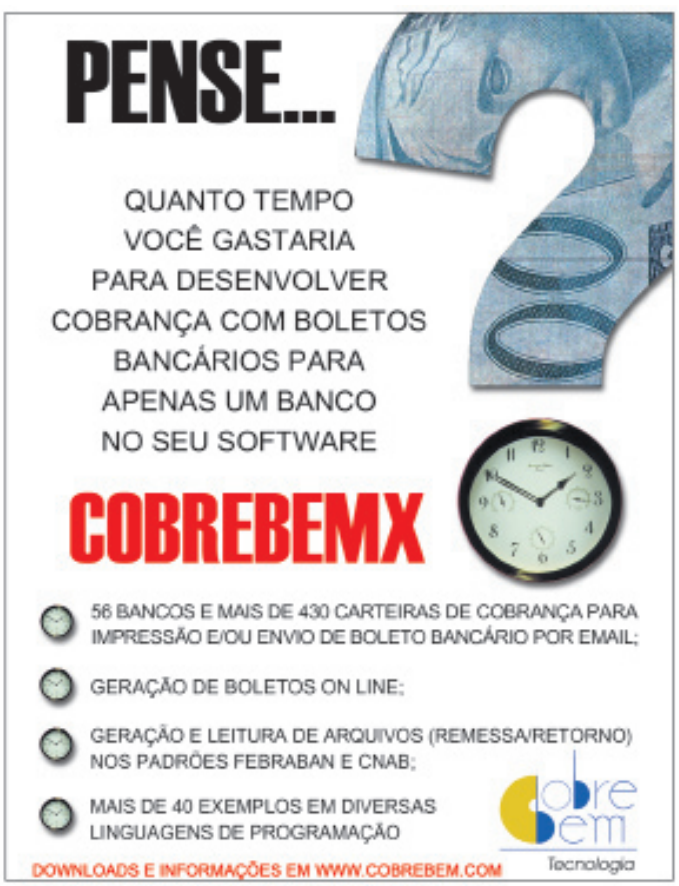
Na **Listagem 8,** é apresentada a abordagem tradicional, sem a declaração de Data Context. Dessa forma, para cada Binding declarado, a mesma fonte é novamente informada. Já é possível prever certo retrabalho caso o TextBox fonte tiver seu nome alterado ou se o vínculo for direcionado a outro controle. Em se tratando de um caso com mais definições de Binding ou caso o elemento em questão possuísse elementos filhos, o nível de retrabalho seria ainda maior.

Listagem 8. Codificação sem a definição de Data Context

```
<ListBox Name="listBoxFontes"
      ItemsSource="{Binding ElementName=textBoxPalavra,
      Path=Text, Mode=OneWay}"
      ToolTip="{Binding ElementName=textBoxPalavra, Path=Text.Length,
      Mode=OneWay}"
      .....
/>
```

Já na **Listagem 9** é apresentada a mesma solução, porém fazendo uso de Data Context. Para passarmos o entendimento da teoria à prática, observe com um pouco mais de cuidado o código apresentado. Veja que para a propriedade DataContext, o contexto definido foi o elemento TextBox, mais precisamente propriedade Text, além do modo One-Way.

Assim sendo, ao se definir o Binding nas propriedades ItemSource e ToolTip do elemento ListBox, é necessário somente que



sejam informadas as particularidades de cada uma, já com base no contexto estipulado. Dessa forma, ItemsSource apenas reaproveita o contexto definido, recebendo assim somente a marcação de Binding. Já em ToolTip, é especificado em Path o valor específico que se deseja dentro do contexto.

Na **Figura 6** temos a aplicação de exemplo em execução, mostrando dados diferentes, em propriedades diferentes, a partir de um mesmo contexto.

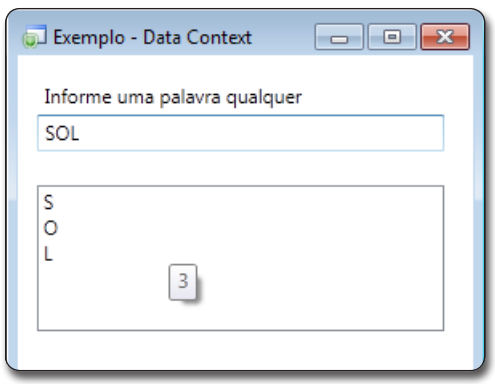


Figura 6. Aplicação de exemplo para Data Context em execução

Listagem 9. Codificação com a definição de Data Context

```
<ListBox Name="listBoxPalavra"
      DataContext="{Binding ElementName=textBoxPalavra,
      Path=Text, Mode=OneWay}"
      ItemsSource="{Binding}"
      ToolTip="{Binding Path=Length}"
      .....
/>
```

Conclusão

Pudemos ver neste artigo mais um recurso presente na tecnologia WPF que, em parceria com o Delphi Prism, fornece meios suficientes para um desenvolvimento tão produtivo e robusto quanto ao que estamos acostumados com o Delphi tradicional.

Ainda sobre este paralelo, aqui mesmo nesta revista foram e continuam sendo publicados artigos que frisam esta aproximação entre Delphi e Visual Studio (Delphi Prism), apresentando comparações entre os IDEs e as linguagens (Object Pascal e Oxygene), visando o aproveitamento de conhecimento. Em suma, conclui-se que a partir do Delphi Prism, uma gama muito grande de opções foi acrescida ao leque do desenvolvedor Delphi.

Em contrapartida, como as possibilidades são muitas, cabe a você mensurar o tipo de projeto e a tecnologia ideal para o seu contexto, e não se deixar levar por modismos. Deixo como desafio, ou tarefa (lembrando os tempos de escola), o aprofundamento do assunto na própria página oficial dos produtos, onde é encontrada uma gama enorme de recursos de forma gratuita. Só para citar, na seção Learn do site do Windows Presentation Foundation, é disponibilizada uma série de materiais técnicos, como artigos, vídeos e outras recomendações, abordando todas vertentes do WPF. Da mesma forma, no site do Delphi Prism, apesar de em menor quantidade, também são encontrados materiais técnicos oficiais.

Agradeço a todos e até a próxima.

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ. Atua como Analista Programador Delphi há 5 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas além de participar do programa Beta Tester Delphi Prism, da Embarcadero.

Delphi Prism XE - Página de apresentação do produto
http://www.embarcadero.com/products/delphi-prism

Delphi Prism XE – Página para download
https://downloads.embarcadero.com/free/delphi_prism

Microsoft WPF - Página oficial
http://windowsclient.net

Microsoft WPF – Learn (Aprendizagem e estudo)
http://windowsclient.net/learn

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Bancos de Dados Gratuitos – Parte 3

Vitor Luiz Rubio

A maneira como os diversos sistemas gerenciadores de bancos de dados são construídos e como os dados são armazenados diferem entre si, assim como a maneira como os softwares se comunicam com eles. Além disso, algumas peculiaridades dos comandos SQL podem mudar de acordo com o fabricante do SGBD.

Geralmente os bancos de dados abrem alguma porta no servidor para que aplicações clientes possam se conectar via TCP/IP e se comunicar com o SGBD. A maneira como essa comunicação é implementada depende muito do fabricante, mas as regras de comunicação são publicadas para que empresas possam desenvolver drivers que acessem esses bancos.

As empresas que desenvolvem drivers expõem publicamente as interfaces/APIs desses drivers para que eles possam ser usados em diversas linguagens de programação, através de bibliotecas e componentes. O Delphi possui, além do DBExpress, apelidado de DBX, o ADO e outros mecanismos. Neste artigo serão explorados alguns deles.

ZeosLib

ZeosLib é um conjunto de bibliotecas para acesso a bancos de dados que permite acesso a uma grande variedade de fabricantes. É open-source, sob licença LGPL, que é livre como a licença GPL, mas não obriga que um software derivado ou que a use seja livre também.

É uma camada de acesso a dados realmente fina e que fornece grande performance. Tem sido a preferida dos desenvolvedores que precisavam de uma alternativa ao DBX, ou em versões anteriores do Delphi que não possuíam DBX. A suíte de componentes ZEOS é muito conhecida dos programadores Delphi 7 e Lazarus. A versão "estável" do pacote ZEOS, a 6.6.6, é compatível apenas com Delphi 2006.

O ZEOS 7.0.0 é ALPHA, por isso deve ser usado com muito cuidado. Ele é compatível com o Delphi 2010, mas pode ser instalado no Delphi XE com algumas modificações.

Para se compilar no Delphi 2010 essas alterações podem ser ignoradas, bastando apenas abrir o arquivo ZeosDbo.groupproj e compilando todos os pacotes, instalando por último o ZComponentDesign150.bpl, como mostra a **Figura 1**.

A **Listagem 1** mostra o arquivo Zeos.inc, na **Listagem 2** está o mesmo arquivo modificado. Isso irá definir algumas diretivas de ambiente que dirão que o Delphi XE é a versão Delphi15, por isso superior às versões 14,12 e assim por diante, e ainda definirá as constantes BDS8, já que o RAD Studio XE é o "BDS 8". Vale

Resumo DevMan

De que trata o artigo:

Este artigo finaliza a série sobre bancos de dados gratuitos trazendo ao leitor mais dois bancos de dados: o MaxDB e o SQLite. Além disso, são apresentados outros componentes interessantes de acesso a bancos de dados.

Em que situação o tema é útil:

O banco de dados a ser escolhido em um projeto depende muito do tipo e tamanho do projeto e da empresa, qual será a carga esperada e proveniente de onde. O programador precisa incrementar sua caixa de ferramentas para conhecer várias alternativas. O MaxDB é o banco de dados do SAP, um dos mais respeitados ERPs do mercado. Uma aplicação que se integre com o SAP, por exemplo um e-commerce, poderia se comunicar diretamente com o banco de dados do SAP em vez de trocar arquivos texto ou XML. A integração poderia ser online em vez de "batch" em um horário pré-determinado. O SQLite é útil em situações que precisam de um banco de dados local para uma aplicação desktop standalone, mas não se deseja usar Access.

Bancos de dados gratuitos:

Além dos bancos de dados "Express" vistos no artigo anterior, existem muitos mais. Neste artigo serão mostrados principalmente do MaxDB e do SQLite. DBX, IBX, ADO e Unidac não são os únicos componentes que existem para se acessar bancos de dados. Neste artigo se apresentará ao leitor os componentes UIB e Zeos e mais algumas dicas e testes serão mostrados. Por fim o artigo será concluído com exemplos de inserção em banco de dados e clonagem de ClientDatasets.

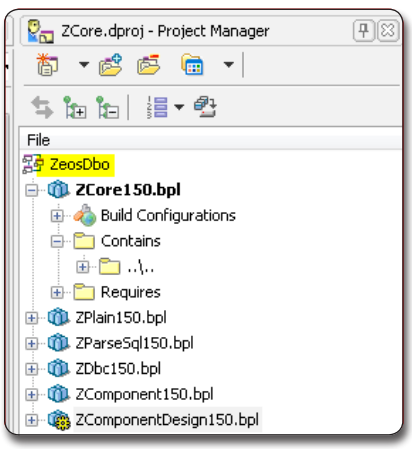


Figura 1. Project Group do Zeos

lembrar que na edição 2005, antes de a Borland se tornar CodeGear, os produtos da Borland Delphi e C++ builder passaram a ser vendidos juntos, com um IDE integrado, multi-linguagem, que passou a se chamar Borland Developer Studio. Daí a razão do nome do produto ser BDS, a versão ser 8 e o executável do Delphi se chamar bds.exe.

Listagem 1. Diretivas para o Delphi 2010

```
// Compilation directives for Delphi 2010
{$IFDEF VER210}
 {$DEFINE VER200BELOW}
 {$DEFINE DELPHI14}
 {$DEFINE DELPHI9_UP}
 {$DEFINE DELPHI10_UP}
 {$DEFINE DELPHI12_UP}
 {$DEFINE DELPHI14_UP}
 {$DEFINE BDS}
 {$DEFINE BDS7}
 {$DEFINE BDS4_UP}
 {$DEFINE BDS5_UP}
 {$DEFINE BDS7_UP}
{$ENDIF}
```

Listagem 2. Diretivas para o Delphi XE

```
// Compilation directives for Delphi XE
{$IFDEF VER220}
 {$DEFINE VER210BELOW}
 {$DEFINE DELPHI15}
 {$DEFINE DELPHI9_UP}
 {$DEFINE DELPHI10_UP}
 {$DEFINE DELPHI12_UP}
 {$DEFINE DELPHI14_UP}
 {$DEFINE DELPHI15_UP}
 {$DEFINE BDS}
 {$DEFINE BDS8}
 {$DEFINE BDS4_UP}
 {$DEFINE BDS5_UP}
 {$DEFINE BDS7_UP}
 {$DEFINE BDS8_UP}
{$ENDIF}
```

No final da edição, o arquivo .inc ficará com os dois conjuntos de diretivas. Algumas referências a VER210, a versão do Delphi 2010, devem ser copiadas sob um ifdef VER220, para que possa ser compilado e instalado no Delphi XE. Deve ser criada uma pasta para Delphi15 na pasta packages, que pode ser copiada da Delphi14.

Em todos os pacotes no projectgroup, nas propriedades do pacote, na guia description, LIB suffix deve ser 150, pois o Delphi XE é a versão 15 (**Figura 2**).

Outra alteração importante para que a suíte ZEOS seja compilada no Delphi XE é no arquivo ZAbstractRODataset.pas. Entre as linhas 3488 e 3512 pode-se encontrar, na versão original, o código da **Listagem 3**, que define os fieldtypes compatíveis com o Delphi 2010 (VER210 definida).

Intercalando esse código com mais uma diretiva de compilação condicional, desta vez fazendo referência às definições de BaseFieldTypes para o Delphi XE e colocando no final mais uma diretiva {$ENDIF} para fechar a nova diretiva condicional, tem-se o código da **Listagem 4**.

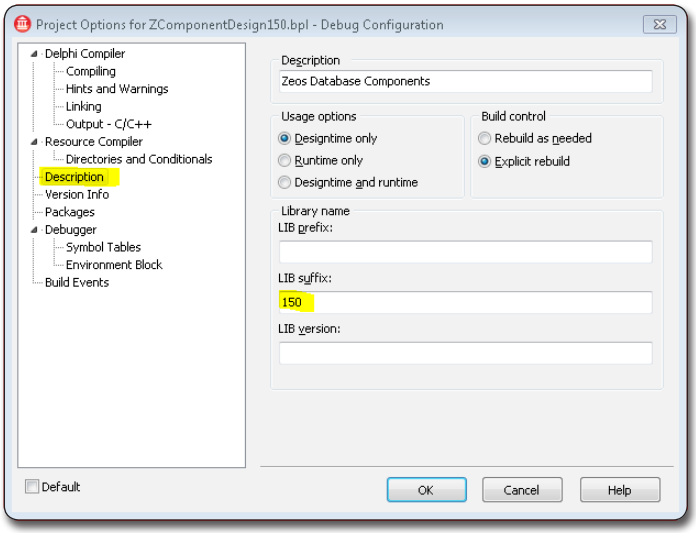


Figura 2. Atributo LIB suffix da propriedade Description dos pacotes

Listagem 3. Diretivas para o Delphi 2010 no arquivo original

```
{$IFDEF VER210}
const
  BaseFieldTypes: array[TFieldType] of TFieldType = (
    ftUnknown, ftString, ftSmallint, ftInteger, ftWord,
    ftBoolean, ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime,
    ftBytes, ftVarBytes, ftAutoInc, ftBlob, ftMemo, ftGraphic, ftFmtMemo,
    ftParadoxOle, ftDBaseOle, ftTypedBinary, ftCursor, ftFixedChar, ftWideString,
    ftLargeint, ftADT, ftArray, ftReference, ftDataSet, ftOraBlob, ftOraClob,
    ftVariant, ftInterface, ftIDispatch, ftGuid, ftTimeStamp, ftFMTBcd,
    ftFixedWideChar, ftWideMemo, ftOraTimeStamp, ftOraInterval,
    ftLongWord, ftShortint, ftByte, ftExtended, ftConnection, ftParams, ftStream,
    ftTimeStampOffset, ftObject, ftSingle);
{$ELSE}

 const
  BaseFieldTypes: array[TFieldType] of TFieldType = (
    ftUnknown, ftString, ftInteger, ftInteger, ftInteger, ftBoolean, ftFloat,
    ftFloat, ftBCD, ftDateTime, ftDateTime, ftDateTime, ftBytes, ftVarBytes,
    ftInteger, ftBlob, ftBlob, ftBlob, ftBlob, ftBlob, ftBlob, ftBlob, ftUnknown,
    ftString, ftString, ftLargeInt, ftADT, ftArray, ftReference, ftDataSet,
    ftBlob, ftBlob, ftVariant, ftInterface, ftInterface, ftString, ftTimestamp, ftFMTBcd);
 {$ENDIF}
{$ENDIF}
{$ENDIF}
{$ENDIF}
```

Como é um pouco complicado compilar a versão 7.0.0 no Delphi XE, juntamente com o fonte desta edição encontra-se compactada a versão 7.0.0 do ZEOS modificada para ser compilada e instalada no XE. Não se esqueça de adicionar os diretórios dos packages e src no search path e browsing path como mostra na **Figura 3**.

A suíte de componentes Zeos pode ser usada para se conectar a muitos bancos de dados diferentes, incluindo aí MSSQL Server e Firebird. Por isso é recomendável que o leitor "revisite" os exemplos anteriores e tente se conectar com Zeos.

MaxDB

O SAPDB mudou de nome para SAP MaxDB, ou simplesmente MaxDB. Ele é o banco de dados do sistema SAP que há algum

tempo foi liberado de maneira open source, mas voltou a ser comercial. Trata-se de um banco de dados bastante robusto e muito usado em empresas que adotam o SAP.

Listagem 4. Diretivas modificadas para o Delphi XE

```
{$IFDEF VER210}
const
  BaseFieldTypes: array[TFieldType] of TFieldType = (
    ftUnknown, ftString, ftSmallint, ftInteger, ftWord,
    ftBoolean, ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime,
    ftBytes, ftVarBytes, ftAutoInc, ftBlob, ftMemo, ftGraphic, ftFmtMemo,
    ftParadoxOle, ftDBaseOle, ftTypedBinary, ftCursor, ftFixedChar, ftWideString,
    ftLargeint, ftADT, ftArray, ftReference, ftDataSet, ftOraBlob, ftOraClob,
    ftVariant, ftInterface, ftIDispatch, ftGuid, ftTimeStamp, ftFMTBcd,
    ftFixedWideChar, ftWideMemo, ftOraTimeStamp, ftOraInterval,
    ftLongWord, ftShortint, ftByte, ftExtended, ftConnection, ftParams, ftStream,
    ftTimeStampOffset, ftObject, ftSingle);
{$ELSE}

{$IFDEF VER220}
const
  BaseFieldTypes: array[TFieldType] of TFieldType = (
    ftUnknown, ftString, ftSmallint, ftInteger, ftWord,
    ftBoolean, ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime,
    ftBytes, ftVarBytes, ftAutoInc, ftBlob, ftMemo, ftGraphic, ftFmtMemo,
    ftParadoxOle, ftDBaseOle, ftTypedBinary, ftCursor, ftFixedChar, ftWideString,
    ftLargeint, ftADT, ftArray, ftReference, ftDataSet, ftOraBlob, ftOraClob,
    ftVariant, ftInterface, ftIDispatch, ftGuid, ftTimeStamp, ftFMTBcd,
    ftFixedWideChar, ftWideMemo, ftOraTimeStamp, ftOraInterval,
    ftLongWord, ftShortint, ftByte, ftExtended, ftConnection, ftParams, ftStream,
    ftTimeStampOffset, ftObject, ftSingle);
{$ELSE}

 const
  BaseFieldTypes: array[TFieldType] of TFieldType = (
    ftUnknown, ftString, ftInteger, ftInteger, ftInteger, ftBoolean, ftFloat,
    ftFloat, ftBCD, ftDateTime, ftDateTime, ftDateTime, ftBytes, ftVarBytes,
    ftInteger, ftBlob, ftBlob, ftBlob, ftBlob, ftBlob, ftBlob, ftBlob, ftUnknown,
    ftString, ftString, ftLargeInt, ftADT, ftArray, ftReference, ftDataSet,
    ftBlob, ftBlob, ftVariant, ftInterface, ftInterface, ftString, ftTimestamp, ftFMTBcd);
 {$ENDIF}
{$ENDIF}
{$ENDIF}
{$ENDIF}
```

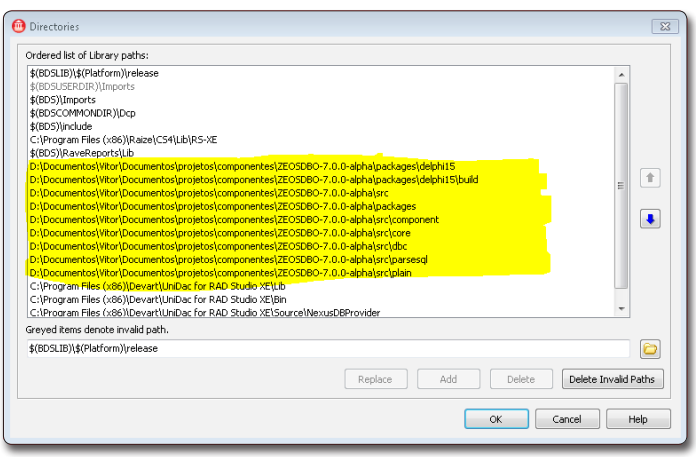


Figura 3. Diretórios dos sources do ZeosDBO

O MaxDB foi criado na década de 80, quando teve vários "nomes comerciais", incluindo a época em que se chamou "Adabas" e era um produto comercial, mudando de nome diversas vezes novamente até assumir o nome de SAP DB.

Ele foi concebido para ser uma alternativa ao Oracle, SQL Server ou DB2. No ano 2000 a SAP liberou o código fonte do SAPDB sob licença GNU GPL, pública e livre.

Em 2003 a SAP juntou forças com o MySql para que pudesse ser desenvolvido um "MySql" comercial mais robusto, sem as limitações impostas pelo projeto GPL. Isso deu origem ao MaxDB. Infelizmente após o MaxDB voltar a ser administrado pela SAP em outubro de 2007 o MaxDB voltou a ser comercial e de código fonte fechado.

Mesmo não sendo mais open source existe uma versão "community" que pode ser baixada e usada gratuitamente, com limitações. É essa a versão usada neste artigo. Sendo assim, as versões 7.2 a 7.4 do SAP DB, referente à época em que ele ainda se chamava SAP DB, são open source pela licença GNU. As versões 7.6 a 7.8 são comerciais e não disponibilizam seu código fonte.

A instalação do MaxDB é um pouco mais complicada que os outros bancos de dados. Porém as versões 7.6, 7.7 e 7.8 são muito mais fáceis de se instalar do que a antecessora 7.4.

Neste artigo será usada a versão 7.8 32 bits. Existe também a versão 64 bit, por isso atenção porque a versão 32 bit não funciona em sistemas 64.

O SGBD não vem com nenhum client ou administrador. Mas há, na mesma página onde se faz o download do banco, links para download de ferramentas gráficas semelhantes ao IBConsole, bastante úteis para se administrar o banco. Baixe e instale os programas DBMgui e SQLStudio para administrar o Max DB, criar tabelas etc.

Também será necessário instalar o driver ODBC para o Max DB. Na seção de links desse artigo há o link principal de download do sistema, onde todo o restante do material pode ser encontrado.

Para iniciar a instalação é escolhida a forma customizada. Um cuidado muito importante durante a instalação do sistema é que as senhas dos usuários DBM e DBADMIN devem ser "secret" (**Figura 4**).

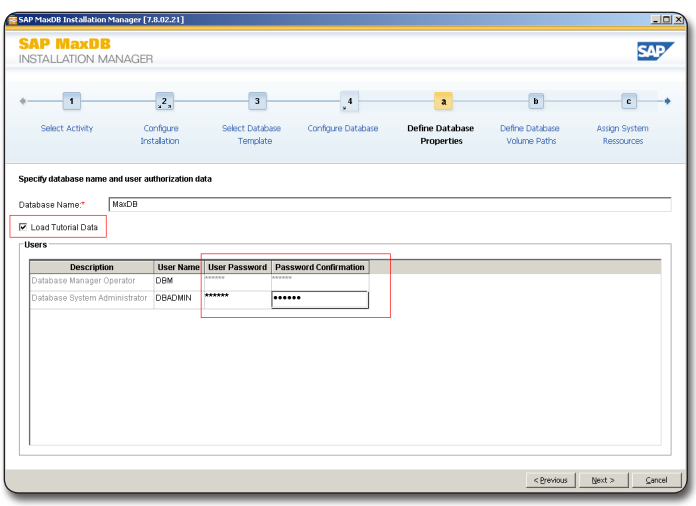


Figura 4. Marcar a opção para criar o "tutorial" e definir as senhas dos usuários de sistema como SECRET

Isto feito deve-se instalar as ferramentas gráficas de administração do MaxDB. Não existe driver DBX nativo e gratuito para este banco, mas é possível se conectar a ele via ADO, BDE ou qualquer outro mecanismo que acesse ODBC.

Ao abrir o programa em MaxDB é possível registrar (**Figura 5**) o banco dados, também é possível clicar em Create e criar um novo banco de dados. Ao criar um novo banco de dados (**Figura 6**) ele aparecerá na lista de bancos de dados registrados. Use sempre username "dbadmin" e password "secret".

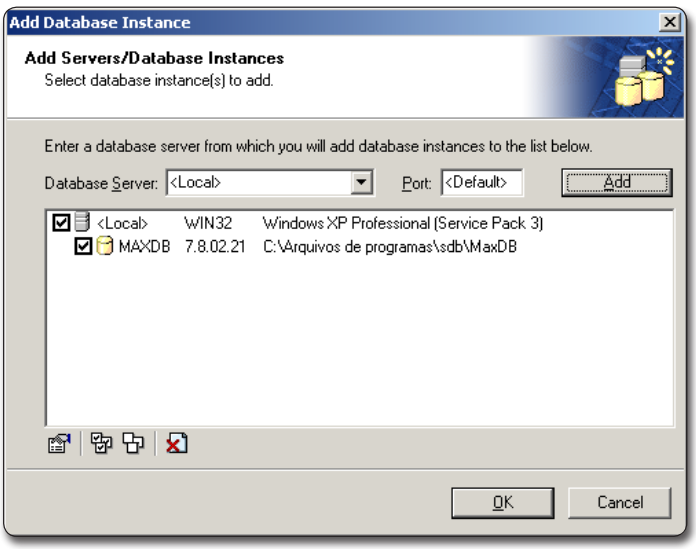


Figura 5. Bancos a serem registrados

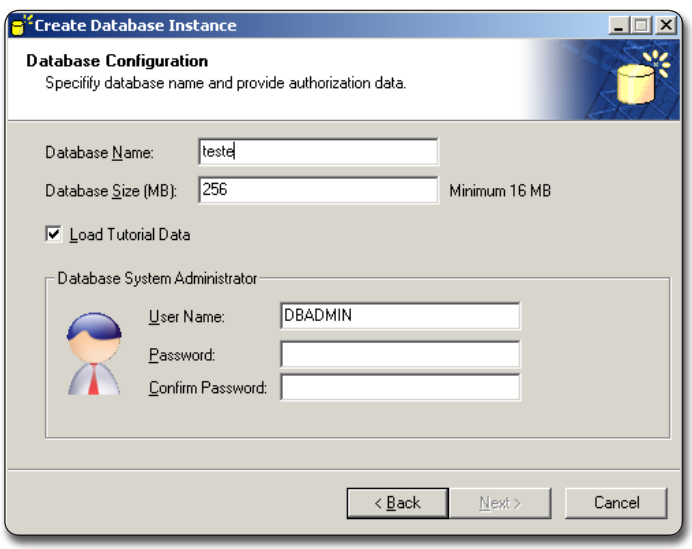


Figura 6. Criação de um novo banco

Depois de se criar um banco de dados de teste é hora de criar algumas tabelas e popular este banco. Através do programa MaxDB>SQL Studio, informamos na propriedade Server que desejamos nos conectar em localhost (ou IP do servidor onde MaxDB está instalado). O Database é MaxDB, ou qualquer outro que se tenha criado. Conectamos usando o login dbadmin e a senha secret. Então surge a tela da **Figura 7**, onde é possível inserir instruções SQL.

As instruções SQL executadas são as da **Listagem 5**, para criação das tabelas contas e logins. Ao final da listagem há instruções insert para população de alguns dados de teste. O SQL Studio permite a execução de apenas uma instrução por vez. É possível selecionar uma instrução para executar somente a que estiver selecionada, nos mesmos moldes do SQL Server Management Studio do MS SQL Server.

Listagem 5. Tabelas contas e logins

```
create table contas
(
  cpf varchar(11) not null primary key,
  nome varchar(50) not null,
  datanascimento timestamp
)

create table logins
(
  cpf varchar(11)  not null ,
  email varchar(50) not null,
  senha varchar(20) not null,
   foreign key fk_cpf (cpf)  references contas (cpf)
)

insert into contas values ('11111111111', 'Vitor Rubio', '1983-02-14 00:00:00.000')
insert into contas values ('22222222222', 'Niklaus Wirth', '1934-02-15 00:00:00.000')
insert into contas values ('33333333333', 'Bjarne Stroustrup', '1950-06-11 00:00:00.000')
insert into contas values ('44444444444', 'Edsger Dijkstra', '1930-05-11 00:00:00.000')
insert into contas values ('55555555555', 'Andrew S. Tanenbaum', '1944-03-16
00:00:00.000')

insert into logins values ('11111111111', 'vitorrubio@gmail.com', 123456)
insert into logins values  ('22222222222', 'contact@niklauswirth.com', 123456)
insert into logins values  ('33333333333', 'contact@bjarnestroustrup.com', 123456)
insert into logins values  ('44444444444', 'contact@edsgerdijkstra.com', 123456)
insert into logins values  ('55555555555', 'contato@andrewstanembaum.com', 123456)
insert into logins values  ('11111111111', 'contato@vitorrubio.com.br', 123456)
```

O script SQL montado representa um fragmento de um e-commerce ou de um sistema qualquer online que atenda várias lojas de uma mesma empresa ou várias empresas de um mesmo grupo. Imagine por exemplo uma livraria que tem lojas físicas, várias lojas virtuais diferentes segmentadas por tema e lojas afiliadas a universidades, por exemplo. Uma mesma pessoa pode se cadastrar em uma loja com um login, mas pode querer se cadastrar em outra loja com um login diferente.

A questão é que mesmo que o usuário tenha múltiplos logins (pares e-mail e senha) no sistema como um todo, ele tem apenas um nome e um CPF. Ou seja, é apenas uma pessoa. Sendo assim nesse sistema hipotético a entidade pessoa foi dividida em duas entidades: conta, que é a pessoa em si, que possui um CPF único, um nome e uma data de nascimento; e logins, já que é possível que tenha múltiplos e-mails e compre em múltiplas lojas do mesmo grupo.

A relação entre contas e logins é de 1 x N, e caso a pessoa tenha em mente se "cadastrar" novamente o seu CPF e nome deve permanecer o mesmo, sem mensagem de erro informando "CPF já cadastrado", mas sim "adicionando" ou "anexando" o novo login à conta/pessoa já existente.

Esse sistema funciona também para casos de comércio onde uma única empresa (CNPJ) pode possuir vários compradores, onde cada um tem seu login e senha. Por questões de simplicidade o endereço foi omitido, e o código da loja não foi incluído na tabela de login.

Também visando a simplicidade optou-se neste exemplo por usar o CPF como chave primária. Em muitos casos isso é possível, em outros é mais recomendável criar um campo inteiro para ser a chave primária e colocar um generator, identity, sequence ou o mecanismo de "autoincremento" que for suportado pelo banco de dados escolhido.

Deve-se ter em mente que campos com "autoincremento" são diferentes de SGBD para SGBD quanto à maneira de se criar. Outros tipos de dados especiais também diferem de acordo com os vários "dialetos" existentes do SQL. A tabela simples criada na **Listagem 5** tendo por chave primária o CPF funcionará na maioria dos bancos de dados com pouca ou nenhuma alteração, o que dá ao leitor a oportunidade de manter um catálogo de exemplos práticos de acesso a diversos bancos de dados.

Alguns dados fictícios foram inseridos nestas tabelas para que se possa executar uma instrução select e trazer alguns dados. Nestes testes um dos usuários possui, como previsto, dois logins.

Sinta-se à vontade para escolher de qual tabela fará o select ou até mesmo usando as duas com "nested datasets".

Conectar-se ao MaxDB é fácil, mas antes de tudo requer uma conexão ODBC. No painel de controle, ferramentas administrativas, fontes de dados ODBC é possível criar uma conexão ODBC com o banco de dados teste criado no MAXDB.

Clicando em adicionar, como mostra a **Figura 8** escolhe-se o driver MAXDB Unicode (não é necessário o Unicode se a sua versão do Delphi não suportar ou se você tentar fazer uma conexão via BDE).

Após clicar em concluir a janela para conclusão de conexão irá surgir. O campo Data Source Name é o nome da fonte de dados ODBC, que aparecerá nas listas de fontes de dados sempre que se tentar fazer uma conexão ODBC com qualquer componente.

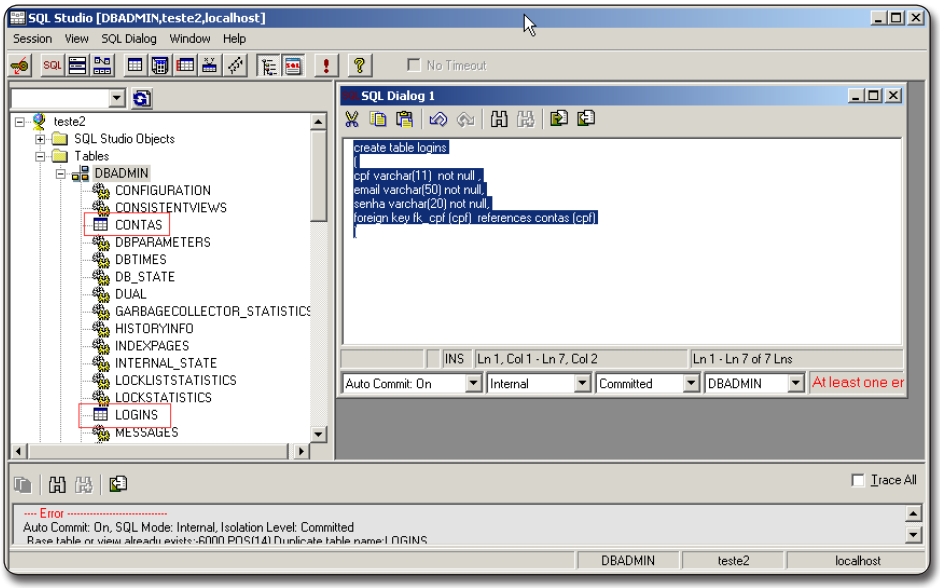


Figura 7. Janela para execução de instruções SQL

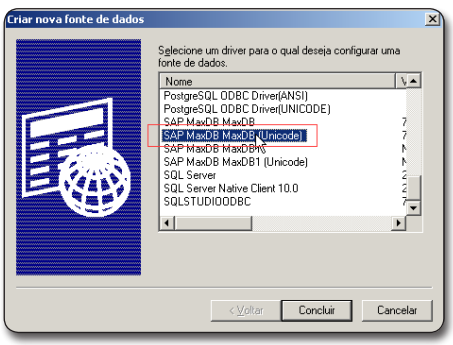


Figura 8. Escolhendo driver MAXDB Unicode

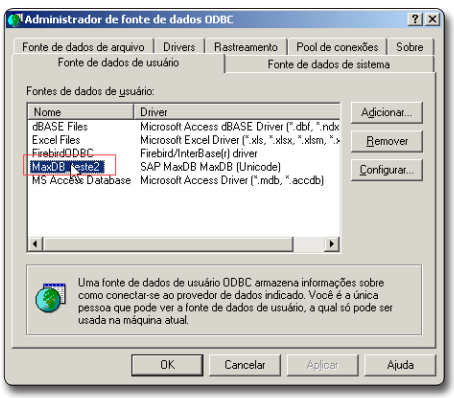


Figura 9. Mostrando a conexão criada

Server é o servidor onde o MaxDB está instalado e Database é o nome do banco de dados criado. Clicando-se no botão Connect é exibida a tela que solicita a entrada do usuário e senha, respectivamente dbadmin e "secret". A **Figura 9** mostra o final do processo com a fonte de dados criada.

Para se conectar via ADO ao MaxDB é necessário colocar um componente TADOConnection e um TADODataset. A propriedade Connection do AdoDataset deve ser a TAdoConnection colocada e sua propriedade CommandText deve ser "select * from contas" ou mais corretamente "select <nomes de campos separados por vírgula> from contas".

Com um duplo clique sobre o componente AdoConnection é possível escolher o driver e o nome da conexão definido no gerenciador de fontes de dados, conforme exibido na **Figura 10**.

Nota

Este mesmo banco de dados da **Listagem 5** será usado nos testes com SQLite. Por motivos de simplicidade o CPF não está sendo validado, mas está sendo considerado como uma string de até 11 caracteres. Para validar o CPF existe um algoritmo chamado "módulo 11" que calculará os dois últimos dígitos com base nos nove primeiros para verificar se o CPF é válido. Neste artigo, como o CPF é totalmente fictício, não está sendo validado. Pesquise em sua ferramenta de busca preferida ou no site da DevMedia por um algoritmo real para validação de CPF. Em um sistema real isso é imprescindível.

Nota

Também é possível criar conexões ODBC por meio de arquivos DSN, que são arquivos de texto muito parecidos com o formato INI que possuem informações detalhadas para a conexão, como nome do driver, servidor, banco de dados, login e password.

Nota do DevMan

Acostume-se a colocar os nomes dos campos no select, mesmo que planeje trazer todos os campos, ainda que os campos sejam muitos. Num sistema bem planejado torna-se rara a necessidade de se trazer todos os campos em uma consulta ou listagem, a não ser na hora de trazer um registro para edição. Tabelas com muitos campos podem ser reavaliadas, pois elas podem sofrer uma "partição vertical" e podem ser separadas em duas ou mais tabelas que sejam ligadas em uma relação 1x1 caso existam muitos campos que permitam null e que são preenchidos apenas em casos especiais.

Nota

Nas conexões com o MaxDB, tanto com ADO, Unidac ou Zeos, os nomes de usuário DBADMIN e a senha SECRET devem ser preenchidos sempre em MAIÚSCULAS.

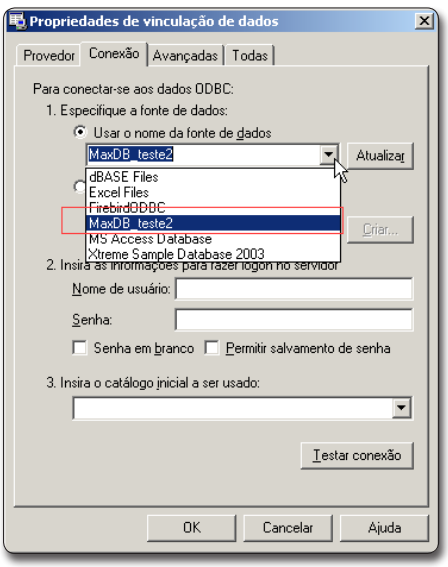


Figura 10. Fonte ODBC criada no passo anterior

Para a conexão com o Unidac é necessário o "quarteto" de componentes UniConnection, UniQuery, UniTransaction e ODBCUniProvider. Um duplo clique no UniConnection abre a janela da **Figura 11**, e a conexão pode ser configurada. A propriedade SQL do UniQuery deve ser "select * from contas"

Por último, depois de instalada a suíte de componentes Zeos é possível encontrá-los na paleta "Zeos Access". Para se conectar ao MaxDB usando Zeos é preciso criar uma conexão ADO, selecionando o protocol ADO conforme mostrado na **Figura 12**. Continuando a configuração do ZConnection, deve-se clicar no botão (...) ao lado da propriedade Database.

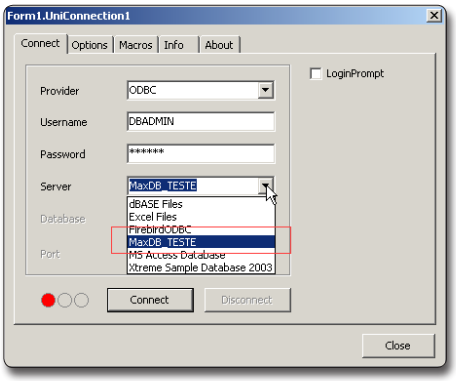


Figura 11. Configuração do Unidac

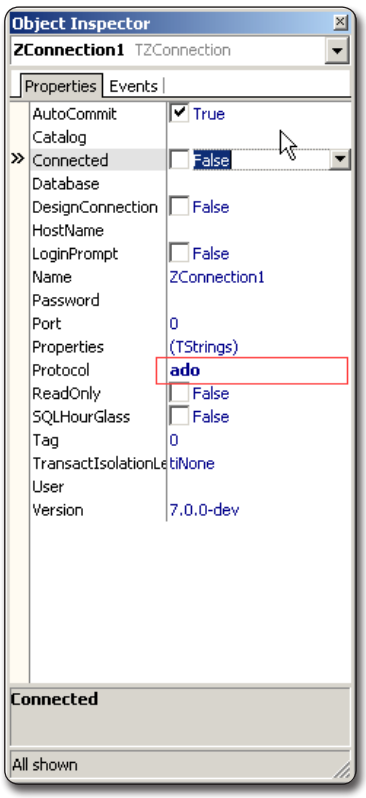


Figura 12. Configuração do Zeos no Object Inspector

Isso fará com que apareça uma tela de conexão ADO. O modo de se conectar também é o mesmo: usa-se o provider OLEDB para ODBC e dentre as conexões ODBC escolhe-se a MaxDB_TESTE, ou o nome colocado durante a definição desta fonte ODBC.

O acesso com ADO se dá exatamente como acontece com os componentes da paleta DBGO, AdoConnection e AdoDataset. A conexão tem como intermediários um provider OLEDB que acessa o banco de dados via ODBC. Quanto mais níveis de acesso, pontes ou gateways, mais lenta seria a conexão. Porém isso ilustra que é possível acessar qualquer banco de dados mesmo que não se tenha um componente específico só para ele. A string de conexão ADO ficará salva na propriedade database do ZConnection.

Adicionando um componente ZQuery e ligando-o ao ZConnection pela propriedade Connection, tem-se o dataset que executará a instrução select e trará os dados. A propriedade SQL do ZQuery deve conter a instrução SQL adequada.

A **Listagem 6** mostra o código para teste e execução do "select" na tabela "contas" do banco de dados TESTE2 do MaxDB usando as três tecnologias citadas: ADO, Unidac e Zeos.

Usa-se neste exemplo os mesmos labels e GetTickCount para medir o tempo de abertura da conexão de cada tecnologia, mas como a tabela usada no teste tem, por enquanto, apenas cinco registros, este teste deve ser completamente descartado, pois cinco registros são insignificantes para se acessar.

Caso em seu ambiente o MaxDB esteja instalado em um servidor, mas o teste esteja sendo desenvolvido em outro, é necessário instalar o driver ODBC do MaxDB. Para isso, depois da janela "Install" do assistente de instalação, desmarque todas as opções de banco de dados e add-ons e deixe marcado apenas a opção de driver ODBC, como mostrado na **Figura 13**.

Existe um driver DBX para ODBC não-oficial, open-source, que pode ser baixado no site do sourceforge. Há na seção de links o endereço para download do mesmo. Mais adiante será mostrado como se instala e usa esse driver.

Tendo a possibilidade de se acessar ODBC com DBExpress torna possível usar DBX para se acessar bancos de dados de aplicações legadas como DBASE e Paradox, além de arquivos Excel e Access, coisa que antes era possível apenas com ADO e Unidac, ou o arcaico BDE.

Driver ODBC para DBX

Instalar esse driver depende se o ambiente é Delphi 2010 ou Delphi XE, e se o sistema operacional é Windows XP ou Windows 7.

Primeiro de tudo é necessário encontrar os arquivos dbxdrivers.ini e dbxconnections.ini. Esses arquivos encontram-se, no caso do Delphi 2010 e dbExpress 3, na pasta <usuário>\Documentos\RAD Studio\dbExpress\7.0.

No caso do Delphi XE eles se encontram em <usuário>\Documentos\RAD Studio\dbExpress\8.0, sendo que nos dois casos <usuário> refere-se ao caminho até a pasta de documentos do usuário partindo da raiz. Este caminho pode mudar de acordo com o sistema operacional.

Descompactando o arquivo dbxoodbc_3_2010_1109alpha1.7z disponível para download na seção de links é possível encontrar o código fonte do driver, juntamente com vários arquivos batch para compilar e alguns binários já compilados. Além disso é possível encontrar alguns arquivos ini pré-configurados.

Da pasta driver\binary\release deve ser copiado o dbxodbc.dll para a pasta onde reside seu exemplo. Para funcionar em runtime copie para a pasta Bin do Delphi também. Se quiser ter informações de debug mais detalhadas caso ocorra uma exception (apenas se você estiver debugando o próprio driver) copie junto os arquivos .map e .jdbg.

Altere o arquivo dbxdrivers.ini do Delphi colocando o conteúdo do arquivo dbx4-drivers.ini na pasta driver\binary do dbxodbc, apresentado na **Listagem 7**.

Tome o cuidado para acrescentar as seções dos arquivos, e não o arquivo inteiro. Mescle acrescentando o conteúdo da seção installed drivers ao final de installed drivers, mas não crie uma seção nova. Apenas a configuração do próprio driver deverá ter uma seção nova. O mesmo se aplica ao dbxconnections.ini.

Abra o arquivo dbxconnections.ini original do Delphi e acrescente no final o conteúdo do arquivo dbx4-connections.ini, apresentado na **Listagem 8**.

Para o Delphi 2010 os arquivos que devem ser usados são os dbx3-*.ini, um para drivers e outro para connections. Depois de copiar a dbxoodbc.dll para o diretório do Delphi deve-se pegar a BPL correspondente, que está na pasta packages\binary e instalá-la no Delphi, da mesma forma que se instala um componente. Se a versão do Delphi usada for a 2010 então o pacote correspondente é o PkgDbxXXDrv2010.bpl, se for o Delphi XE então o pacote é o PkgDbxXXDrv2011.bpl.

Copie a BPL correspondente ao seu Delphi para a pasta Bin do Delphi, depois vá no menu Component>Install packages e procure pela BPL em questão na pasta Bin, depois marque-a, conforme visto na **Figura 14**.

Os arquivos podem ter múltiplas formas de conexão a múltiplos bancos de dados, já que o ODBC pode se conectar a vários bancos

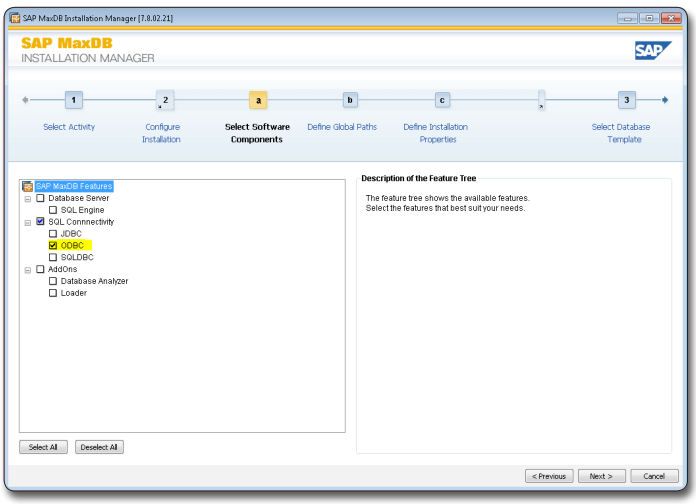


Figura 13. Instalando apenas o Driver ODBC

Listagem 6. Teste de conexão ao MaxDB

```
procedure TForm1.Button1Click(Sender: TObject);
var
 T: integer;
begin

 if ZQuery1.Active then
  ZQuery1.Close;

 t := GetTickCount;
 ZQuery1.Open;
 t := GetTickCount -t;
 Label3.Caption :=  IntToStr(t);
 DataSource1.DataSet := ZQuery1;
end;

procedure TForm1.Button5Click(Sender: TObject);
var
 T: integer;
begin

 if ADODataSet1.Active then
  ADODataSet1.Close;

 t := GetTickCount;
 ADODataSet1.Open;
 t := GetTickCount -t;
 Label1.Caption :=  IntToStr(t);
 DataSource1.DataSet := ADODataSet1;
end;

procedure TForm1.Button6Click(Sender: TObject);
var
 T: integer;
begin

 if UniQuery1.Active then
  UniQuery1.Close;

 t := GetTickCount;
 UniQuery1.Open;
 t := GetTickCount -t;
 Label2.Caption :=  IntToStr(t);
 DataSource1.DataSet := UniQuery1;
end;
```

de dados, tanto através de um registro no administrador de fontes de dados ODBC como através de um arquivo dsn. No final das contas uma simples string pode definir uma conexão com um banco de dados. Essa string deve conter nome do servidor, nome do banco, porta, login, password, driver e outras informações opcionais.

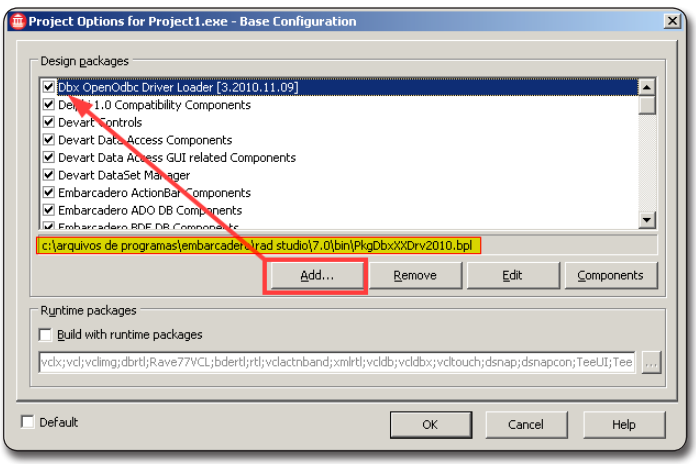


Figura 14. Instalando o pacote

A maneira de se conectar dependerá muito do seu banco de dados, do driver ODBC disponível para ele e de como uma connection string para esse driver é formada. A versão do Delphi 2010 vem com muitos "exemplos" de formas de conexão aos mesmos bancos de dados. Esses exemplos podem não ajudar em nada se você não souber onde e como se conectar. Além disso, os exemplos podem ser repetitivos, mostrando várias maneiras diferentes de se conectar ao mesmo banco de dados. Já a versão do Delphi XE traz apenas o driver "puro".

Seja qual for a versão do Delphi ou o exemplo de conexão que você use, algumas propriedades não mudam:

Database (ou DatabaseName) → é o nome do banco de dados. Coloque DSN=<Nome da fonte de dados no administrador de fontes de dados ODBC>

User (ou Login) → Nome de usuário. No caso do MaxDB é DBADMIN.

Password → pode ser em branco nos casos de conexões com Excel, Dbase ou Paradox.

Connection String → Vai ser formada com parâmetros opcionais e terá automaticamente "concatenada" em seu início as informações de database, user e password.

Listagem 7. Mesclar ao arquivo dbxdrivers.ini

```
[Installed Drivers]

; Dbxoodbc system odbc manager driver:
DbxOpenOdbc=1
;-------------------------------------------------------------------------------
[DbxOpenOdbc]
LibraryName=dbxoodbc.dll
ProductName=DbxOpenODBC
MetaDataPackageLoader=TDBXOpenODBCMetaDataCommandFactory
; @d2007 critical "DriverPackageLoader":
DriverPackageLoader=TDBXDynalinkDriverLoaderOpenOdbc
;
;GetDriverFunc=getSQLDriverODBC ; is customized
;VendorLib=odbc32.dll ; is customized
; On linux platform need replace VendorLib to VendorLibLinux:
;VendorLibLinux=odbc32.so
;OpenOdbc TransIsolation=ReadCommited ; is customized
;Database=
;User_Name=user
;Password=password
;RowsetSize=20
;BlobSize=-1
; Delphi7Up or 'Delphi6 + DbxXXDrv.pas'
;Trim Char=True
;Custom String=coNetPacketSize=8192
;coLockMode=17
;ConnectionString=DSN=MY_DSN;coNetPacketSize=8192
;coLockMode=17;coBlobChunkSize=40960

[DbxOpenOdbc TransIsolation]
DirtyRead=0
ReadCommited=1
RepeatableRead=2
;-------------------------------------------------------------------------------
```

Listagem 8. Mesclar ao arquivo dbxconnections.ini

```
' Version 3.025, 2006-01-12

' For ease of switching connections, you can add connections to dbxconnections.ini
' This is normally located in
' ${DBX.*.ini filder}\dbxconnections.ini
'  Delphi 2010: "%ALLUSERSPROFILE%\Documents\RAD Studio\dbExpress\7.0\
dbxconnections.ini"
'  or see registry path:
'  [HKEY_CURRENT_USER\Software\CodeGear\BDS\7.0\DBExpress]
'
' Examples of connection settings for dbxoodbc Open Odbc Driver...
' Obviously you will need to change them for you own setup
(except for Odbc Prompt - this works as-is)
;-------------------------------------------------------------------------------
[DbxOpenOdbcConnection]
DriverName=DbxOpenOdbc
; is customized:
GetDriverFunc=getSQLDriverODBCW
; is customized:VendorLib=odbc32.dll
;VendorLibLinux=odbc32.soDatabase=?
User_Name=name
Password=secret
BlobSize=-1
RowsetSize=20
OpenOdbc TransIsolation=ReadCommited
; Delphi7Up or 'Delphi6 + DbxXXDrv.pas'
Trim Char=True
;Prepare SQL=False
;Custom String=coNetPacketSize=8192;coLockMode=17
;coBlobChunkSize=40960 ConnectionString=DSN=MY_DSN
;coNetPacketSize=8192
;coLockMode=17;coBlobChunkSize=40960
MetaDataPackageLoader=TDBXOpenODBCMetaDataCommandFactory
; @d2007 critical "DriverPackageLoader":
DriverPackageLoader=TDBXDynalinkDriverLoaderOpenOdbc
;-------------------------------------------------------------------------------
```

A melhor maneira de saber como se conectar com um banco de dados qualquer via ODBC é criar uma "fonte de dados de arquivo" no Administrador de fontes de dados e salvar um arquivo .dsn. Abrindo o arquivo dsn no bloco de notas é possível ver a connection string dele. Deve-se apenas retirar as quebras de linhas e usar ";" no lugar.

Um driver ODBC para o dbExpress te dá uma opção a mais para se conectar em qualquer banco de dados. Se tiver que conectar em uma planilha Excel ou em um banco de dados legado como Dbase ou Paradox é possível usar Unidac, ADO, Zeos e também DBX com esse driver.

Exemplo de connection string para se conectar ao MaxDB via arquivo DSN:

```
FILEDSN=<<caminho do arquivo>\maxdb.dsn
```

O arquivo MaxDB.dsn está disponível junto com os arquivos desta edição. Ele deve ser editado no bloco de notas para que sejam trocadas informações como nome do banco ou nome de usuário, lembrando que nestes exemplos o nome do usuário é DBADMIN e a senha é SECRET tudo em maiúsculas.

A **Figura 15** mostra esta configuração. Neste caso a propriedade LibraryName é dbxoodbc.dll e a VendorLib é odbc32.dll.

Outros componentes que permitem conexão via ODBC, como ADO, Zeos e Unidac também permitirão a conexão via arquivo DSN. Para se conectar direto com uma connection string que especifique o driver, banco de dados etc., a connection string é:

```
DRIVER={SAP MaxDB MaxDB (Unicode)};SERVERNODE=localhost;SERVERDB=teste2;UID=DBADMIN;PWD=SECRET;
```

Estas mesmas regras de formulação de connection strings para conexão via ODBC valem para conexões com ODBC via ADO, Zeos ou Unidac. Tanto no Zeos como no DBX a connection string deve ir na propriedade database. No Unidac a string de conexão vai na propriedade Server. No ADO a string de conexão vai no box "usar sequência de conexão".

Por último é possível conectar-se a qualquer banco de dados que esteja registrado

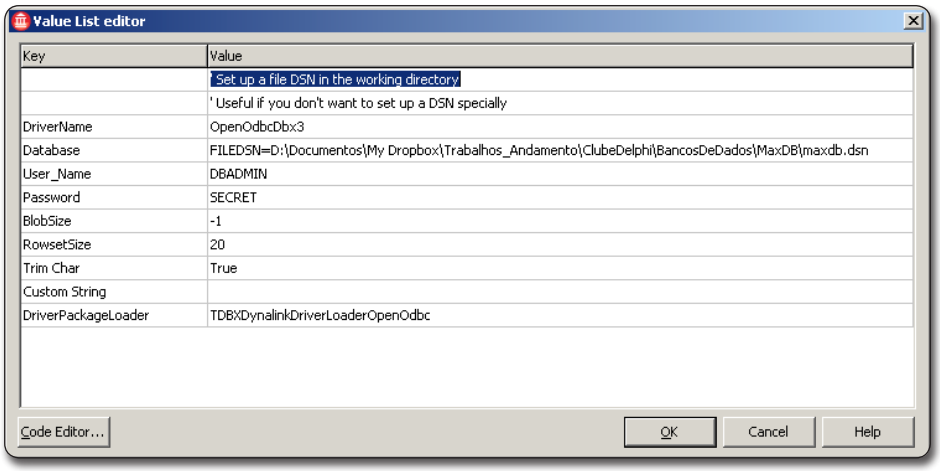


Figura 15. Configurando o DBX + ODBC para acessar o MaxDB

no Administrador de fontes de dados ODBC apenas através de seu nome, colocando na propriedade database a string DSN=MaxDB_TESTE. Isto também vale para Zeos, ADO e Unidac.

O pacote dbxoodbc_3_2010_1109alpha1.7z ainda contém vários arquivos .bat para compilar o driver nas várias versões do Delphi via linha de comando, pacotes bpl com informações de debug para várias versões do Delphi e uma pasta de exemplos de demonstrações com vários bancos de dados.

A distribuição de um sistema que se conecte a um banco de dados qualquer via ODBC depende muito da maneira como a conexão foi feita.

Em todos os casos devem ser distribuídas junto com a aplicação as dlls dbxoodbc.dll e odbc32.dll. Se a conexão for feita através de um arquivo DSN então este deve ser distribuído junto com a aplicação, e modificado de acordo com o ambiente (ip do servidor, nome do banco de dados, usuário, senha etc.).

Se a conexão for feita via connection string, uma string de conexão pode ser armazenada num arquivo INI ou XML, ou mesmo "composta" através de várias propriedades armazenadas em um arquivo INI.

A maneira menos vantajosa, que deve ser usada apenas em casos de uma única máquina ou desenvolvimento, é a maneira em que se conecta direto a um Data Source (DSN) registrado, devido ao trabalho que é registrar o nome da conexão no Administrador de fontes de dados ODBC em cada computador que o sistema rodar.

Nota do DevMan

Existem problemas de compatibilidade entre drivers de 32 e 64 bits, o que gera muitos problemas ao se tentar fazer uma conexão ODBC no Windows 64.

O exemplo de conexão com MaxDB foi entendido para que o leitor pudesse ver todas as formas de conexão via ODBC com Zeos, Ado, DBX e Unidac. São 12 formas de conexão no total. Lembrando que o mesmo vale para outros bancos de dados como Firebird+ODBC, MySQL + ODBC e assim por diante. Levando em consideração que o BDE também pode se conectar a fontes de dados ODBC (contanto que não sejam Unicode) então o leque de opções aumenta.

Além de tudo isso é importante salientar que o ClientDataset se conecta a qualquer tipo ou descendente de Dataset. Podem ocorrer erros durante uma inserção, alteração ou exclusão, porque talvez algum dataset, principalmente os antigos/legados, não implementem corretamente algumas interfaces. Mas para trazer os dados sempre funcionará. Isso quer dizer que não existe desculpa para se usar um ClientDataset como um cache desconectado para se ganhar performance. As alterações depois podem ser aplicadas fora do ClientDataset, direto no dataset de origem que está ligado ao provider.

Nota do DevMan

Algumas suítes de componentes, como o Zeos e o Unidac não possuem um componente do tipo xxxDataSet, mas possuem do tipo xxxQuery (onde xxx é uma referência ao pacote). Esse tipo de componente pode ser ligado a um xxxUpdateObject, da mesma forma como se fazia com BDE ou IBX para se customizar a maneira como serão executados os comandos de update, insert e delete, através de templates.

SQLite

SQLite é uma biblioteca escrita em C, cross-platform, o que significa que ela pode ser compilada e usada em Linux e outros sistemas operacionais que tenham um compilador C. Esta biblioteca serve para criar bancos de dados em um único arquivo (self-contained), sem necessidade de um servidor (serverless) e com configuração zero.

Esta biblioteca é mantida por um consórcio de várias empresas incluindo Oracle, Nokia e Mozilla. Ela tem sido muito usada no mundo mobile, já que a maioria dos sistemas operacionais de celulares é de uma forma ou de outra baseado no Linux, e mesmo que não seja sempre existem compiladores C para a plataforma.

Outra aplicação muito útil para essa biblioteca é em pequenos bancos de dados locais, embarcados, para aplicações. Se você possui Mozilla Firefox instalado no seu computador faça uma busca por *.sqlite. Provavelmente achará arquivos que contenham listas, configurações, preferências e dados de plugins nesses arquivos. Outras aplicações onde ele é usado incluem iPhones, Skype e o antivírus McAfee.

Além das vantagens mencionadas o SQLite é rápido, leve (apenas uma DLL e um arquivo de banco) e totalmente independente de qualquer coisa que esteja instalada no ambiente que rodará o sistema. E é totalmente compatível com a linguagem SQL. Ou seja, dados são gravados, mantidos e deletados via instruções insert, update, delete, e são recuperados via instruções select.

Ele é indicado para todos os cenários de aplicações pequenas que precisam manter listas de opções ou bancos de dados pequenos, locais, como listas de músicas, artistas e álbuns de um player.

Em todos os cenários que o programador escolheria usar um ClientDataset "desconectado" e um arquivo XML pode ser usado o SQLite, que tem a vantagem de armazenar todas as tabelas no mesmo arquivo. Usando os arquivos XML ou CDS do ClientDataset (chamados MyBase) tem-se a desvantagem de ter muitos arquivos, um para cada tabela, e de não ter suporte à instruções SQL.

Nos cenários com bases de dados um pouco mais complexas, onde o programador usaria MS Access para armazenar os dados, o desenvolvedor pode tranquilamente substituir por SQLite com a vantagem de rodar em qualquer computador, sem que exista o Access instalado e sem a necessidade de instaladores de pesadas bibliotecas para acessar o Access, como o Microsoft JET. Não há a necessidade de drivers ODBC, OLEDB ou de se criar fontes de dados ODBC ou desenterrar o BDE.

É muito adequado para as aplicações de "caixinha" vendidas em supermercado que funcionam offline em um único desktop e sem a necessidade de um servidor de bancos de dados. Geralmente nesses casos o programador tem algumas opções além do MS Access, Dbase/DBF, ou paradox, que geram um arquivo por tabela mais os arquivos de índices. Estes mecanismos apresentam dificuldades que vão desde corrupção de arquivos ou índices a perda de dados. O backup também é problemático neste cenário.

Além disso, com exceção do Access, os arquivos DBF e Paradox suportam comandos SQL apenas com a ajuda do BDE ou de um driver ODBC, que fará o papel intermediário de interpretar os comandos SQL e retornar um subconjunto dos dados existentes.

Outro cenário provável de uso do SQLite são pequenos sites onde poucas páginas terão seu conteúdo modificado por um administrador, ou ainda sistemas que têm muitas configurações e muitas opções de personalização. Nesses casos um arquivo INI ou XML ficaria muito grande, com muitas seções, e muito difícil de gerenciar.

O SQLite foi todo escrito com a preocupação de se usar apenas ANSI C puro e não usar nada mais que as funções ou bibliotecas padrão do C. Além disso, o desenvolvedor C ou C++ que desejar pode incluir o código fonte do SQLite (um arquivo .c e um .h) diretamente junto com o source de sua aplicação e compilar junto, embarcando assim o SQLite em sua aplicação e não precisando distribuir uma DLL.

Ao contrário de arquivos XML, Access, DBASE e Paradox o SQLite é transacional, isso significa que cada transação é atômica e que pode ser refletida inteiramente no banco de dados ou descartada inteiramente. Não é permitida "meia transação", isso significa que o arquivo de dados não será corrompido por um travamento do programa, do sistema operacional ou por uma queda de energia.

Nota do DevMan

SQLite tem várias vantagens em relação aos "bancos de dados" serverless ou desktop, como o fato de ser transacional e permitir que vários aplicativos acessem o mesmo banco de dados. Ele também pode ser mais rápido em relação a bancos de dados cliente-servidor, como Firebird ou Oracle, porque não necessita de comunicação interprocessos, como TCP/IP, para funcionar. Ele não tem um SGBD que abre uma porta em um servidor para receber comandos e responder como os outros bancos de dados vistos neste artigo, comunicando-se diretamente com o arquivo. Isso tem suas vantagens, mas também tem desvantagens: uma delas é a segurança, pois qualquer um com acesso ao arquivo tem acesso aos dados. Outra desvantagem é a estabilidade, visto que ele não é imune a bugs nas aplicações clientes. Por último ele não é escalável, servindo como banco de dados cliente-servidor de uma rede. Para simular um ambiente cliente-servidor com ele, com múltiplos clientes, pode-se tentar criar Web Services ou usar DataSnap, onde o servidor acessará o SQLite localmente. Mas isso não é recomendável. Para os cenários onde isso for necessário é melhor usar SGBDs completos.

Para começar a usar o SQLite é necessário entrar no site do projeto e baixar os binários pré-compilados para Windows.

Em desenvolvimento são necessários dois arquivos: a DLL do SQLite (sqlite-dll-win32-x86-3070701.zip) e o Shell para execução de comandos (sqlite-shell-win32-x86-3070701.zip) que será usado para criar as tabelas do exemplo.

Para distribuir a aplicação o shell não será necessário, precisando apenas da sqlite3.dll. O banco de dados deve ser criado através do prompt de comando (cmd). Entre pelo cmd no diretório onde

se encontra o sqlite3.dll e o sqlite3.exe (o Shell) e executando o comando sqlite3 clubedelphi.sqlite onde clubedelphi.sqlite é o nome do banco de dados.

Depois disso os comandos SQL podem ser executados terminados em ";" ou lidos de um arquivo. Crie um arquivo chamado banco.sql com o conteúdo da **Listagem 9** e execute o comando ".read banco.sql" no Shell do SQLite.

Listagem 9. Arquivo banco.sql

```
create table contas
(
  cpf varchar(11) not null primary key,
  nome varchar(50) not null,
  datanascimento timestamp
);

create table logins
(
  cpf varchar(11)  not null,
  email varchar(50) not null,
  senha varchar(20) not null,
  foreign key (cpf)  references contas (cpf)
);

insert into contas values ('11111111111', 'Vitor Rubio', '1983-02-14 00:00:00.000');
insert into contas values ('22222222222', 'Niklaus Wirth', '1934-02-15 00:00:00.000');
insert into contas values ('33333333333', 'Bjarne Stroustrup', '1950-06-11 00:00:00.000');
insert into contas values ('44444444444', 'Edsger Dijkstra', '1930-05-11 00:00:00.000');
insert into contas values ('55555555555', 'Andrew S. Tanenbaum', '1944-03-16
00:00:00.000');

insert into logins values ('11111111111', 'vitorrubio@gmail.com', 123456);
insert into logins values  ('22222222222', 'contact@niklauswirth.com', 123456);
insert into logins values  ('33333333333', 'contact@bjarnestroustrup.com', 123456);
insert into logins values  ('44444444444', 'contact@edsgerdijkstra.com', 123456);
insert into logins values  ('55555555555', 'contato@andrewstanembaum.com',
123456);
insert into logins values  ('11111111111', 'contato@vitorrubio.com.br', 123456);
```

O SQLite não tem suporte a chaves estrangeiras com nome de constraint. Em todos os bancos de dados um SGBD assegura que uma inserção na tabela filha (logins) só seja possível caso exista o CPF correspondente na tabela mãe (contas) se as duas tiverem uma relação de 1:n com uma chave estrangeira em logins referenciando a tabela de contas.

Essa verificação é feita através de "triggers" e "constraints" automáticas que verificam a existência de um registro em outra tabela e disparam uma exceção caso haja violação da integridade referencial.

O SQLite tem suporte a chaves estrangeiras e constraints automáticas, mas apenas se for compilado com algumas opções especiais ativadas. Como nesse artigo está sendo usada a versão pré-compilada (sem essas opções especiais do pré-compilador) a biblioteca usada aqui não dá suporte a essas constraints automáticas.

A consequência disso é que mesmo que tenha-se definido uma chave estrangeira, o SQLite permitirá normalmente que se inclua um login para um CPF que não existe na tabela de contas. Todas as validações e garantias de integridade devem ser feitas pela aplicação nesse caso.

Existem drivers ODBC para SQLite, e existem também providers para .Net, mas neste artigo será mostrado como se conectar de forma nativa ao SQLite usando o Unidac e o Zeos.

Para tornar possível que a conexão seja feita em tempo de design no Delphi, é necessário ter uma cópia da sqlite3.dll na pasta Bin do Delphi ou na pasta system32 do Windows.

Tanto com Zeos como com Unidac a única propriedade que precisa ser preenchida para a conexão é a propriedade Database, que é o caminho físico do arquivo .sqlite. Para se conectar com o Zeos é necessário um zConnection e um zQuery, e a propriedade protocol do zConnection deve ser sqlite-3.

Para se conectar via Unidac é necessário um Uniconnection, UniTransaction, SQLiteUniProvider e um Uniquery. A query usada nos dois componentes é "select * from contas".

O código dos testes até agora tem sido repetitivo, pois não foram levadas em consideração questões como reaproveitamento de código ou princípios da orientação a objetos. O objetivo aqui é mostrar basicamente como acessar os dados.

Rode a aplicação e faça testes de inserção, exclusão e alteração. Também é possível criar conexões via ADO usando para isso um driver ODBC, porém os drivers ODBC para SQLite encontram-se atualmente em estágio beta.

Existe ainda um driver DBX também feito pela Devart com a mesma qualidade do Unidac. Ele permite que conexões ao SQLite sejam feitas via DBExpress.

IBO

A suíte de componentes IBO (IB objects) é bastante diversificada e possui vários componentes, muitos dos quais redundantes ou inviáveis de se usar em uma aplicação que aspire ser multibanco. É que além dos datasets próprios ele também possui controles data-aware próprios, como dbEdits e afins, grids próprias e até datasource próprio. O uso desses componentes é recomendado apenas se sua aplicação usar apenas Interbase ou Firebird sem nenhuma perspectiva ou desejo de migração de banco de dados.

O link para download do trial encontra-se na seção de links deste artigo e após a instalação várias "paletas" novas de componentes serão adicionadas ao IDE do Delphi. A paleta que interessa aos propósitos desse artigo é a IBO TDataset, que possui componentes que seguem o padrão do Delphi.

Adicionando um IBODatabase, IBOTransaction e um IBOQuery e setando a propriedade SQL do IBOQuery para "select * from employees" o restante da conexão pode ser feita seguindo-se os mesmos passos para uma conexão com IBX.

A propriedade database é o caminho do banco de dados, que pode ser precedido de localhost: caso o banco esteja local ou NomeDoServidor: caso esteja em outra máquina. O SQL Dialect é 3 e o user name e password é o padrão do firebird.

O IBO se mostra mais rápido que o IBX, e pouca coisa mais lento que o Unidac, tornando-se uma boa opção para empresas que trabalham somente com Interbase e Firebird. Os componentes têm tantas propriedades a serem configuradas, como questões

de cache, por exemplo, que ganhos de performance consideráveis podem ser conseguidos dependendo do ambiente e das configurações.

Um atrativo interessante do IBO é que ele tem janelas de diálogo e editores de componente muito bem elaborados. Bom para os casos de conectar ou testar a conexão em design-time, utilizando o próprio IDE.

Listagem 9. Arquivo banco.sql

```
create table contas
(
  cpf varchar(11) not null primary key,
  nome varchar(50) not null,
  datanascimento timestamp
);

create table logins
(
  cpf varchar(11)  not null,
  email varchar(50) not null,
  senha varchar(20) not null,
  foreign key (cpf)  references contas (cpf)
);

insert into contas values ('11111111111', 'Vitor Rubio', '1983-02-14 00:00:00.000');
insert into contas values ('22222222222', 'Niklaus Wirth', '1934-02-15 00:00:00.000');
insert into contas values ('33333333333', 'Bjarne Stroustrup', '1950-06-11 00:00:00.000');
insert into contas values ('44444444444', 'Edsger Dijkstra', '1930-05-11 00:00:00.000');
insert into contas values ('55555555555', 'Andrew S. Tanenbaum', '1944-03-16
00:00:00.000');

insert into logins values ('11111111111', 'vitorrubio@gmail.com', 123456);
insert into logins values  ('22222222222', 'contact@niklauswirth.com', 123456);
insert into logins values  ('33333333333', 'contact@bjarnestroustrup.com', 123456);
insert into logins values  ('44444444444', 'contact@edsgerdijkstra.com', 123456);
insert into logins values  ('55555555555', 'contato@andrewstanembaum.com',
123456);
insert into logins values  ('11111111111', 'contato@vitorrubio.com.br', 123456);
```

AnyDac

AnyDac é a suíte de componentes da DA-Soft e é similar ao Zeos ou Unidac: tem o objetivo de ser uma solução única para a conexão com diversos bancos de dados.

Na seção de links existe um para o download da versão Trial do AnyDac. A ideia é que o leitor refaça todos os exemplos feitos nesse mini-curso até agora usando o AnyDac. Como exemplos, nos arquivos que estão sendo enviados com esta edição, algumas conexões foram feitas com Firebird e SQLite.

Dependendo da linguagem ou versão do seu Windows (32 ou 64 bits) o local de instalação das BPLs de runtime e designtime do AnyDac (C:\Program Files (x86)\da-soft\AnyDAC for Delphi\Dcu\D15) deve ser adicionado à variável de ambiente "path" do seu Windows ou então as BPLs deste diretório copiadas para a pasta Bin do Delphi ou para a pasta padrão de output de BPLs.

No quesito usabilidade o AnyDac é muito mais parecido com o DBX do que com o Unidac. Ele adiciona uma série de plug-ins no Delphi (os experts) com um navegador de bancos de dados muito semelhante ao Database Explorer (ou ao arcaico Database Desktop da época do BDE). Só que com a desvantagem de que não dá para arrastar uma conexão para um formulário para criar uma nova.

Ao se conectar a um banco de dados qualquer é possível escolher entre um nome de conexão pré-definida, assim como no DBX com a propriedade connection name, ou construir uma conexão editando manualmente todos os parâmetros.

Com dois cliques no componente ADConnection é possível visualizar a janela para configuração da conexão, mostrada na **Figura 16**. Em destaque encontra-se o nome do driver, no caso SQLite, e o endereço do banco de dados.

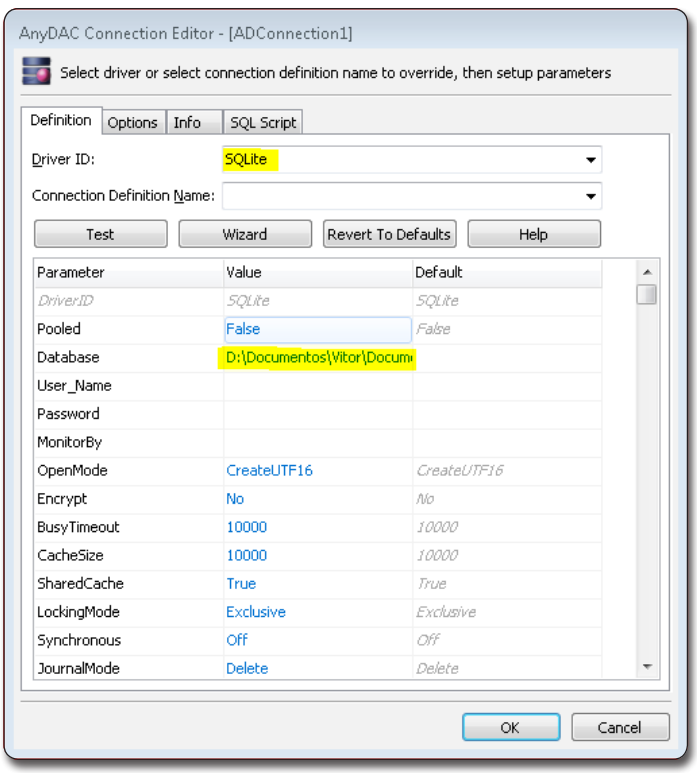


Figura 16. Edição do ADConnection

Uma conexão com o AnyDac precisa de um ADConnection, um ADTransaction, um ADQuery (paleta AnyDac), um ADGUIxWaitCursor (encontra-se na paleta AnyDac UI e serve para registrar o cursor de espera automaticamente) e um ASPhis***DriverLink, onde *** é o nome do driver de conexão ao banco de dados, como ADPhys**SQLite**DriverLink1 ou ADPhys**IB**DriverLink1. Esses se encontram na paleta AnyDac Links.

Outro recurso interessante do AnyDac é o Query Builder para editar a propriedade SQL dos ADQuery, que mostra graficamente tabelas, campos e relações de uma maneira parecida com o MS Access. Além disso é possível executar a query, testar, salvar ou carregar o SQL a partir de um arquivo usando essa janela (**Figura 17**).

Um ponto interessante do AnyDac é que a connection (e as ADQueries) podem aceitar dois componentes transaction: um somente para os selects/consultas e outro para os updates (propriedades Transaction e UpdateTransaction respectivamente). O IBO e outras suítes de componentes também apresentam

comportamento similar. A mesma transaction pode ser ligada às duas propriedades sem problema algum. Caso a propriedade UpdateTransaction não seja preenchida então o componente não permitirá inserções, atualizações ou exclusões, tornando-se somente leitura.

O componente ASQuery também aceita um TADUpdateSql para instruções SQL customizadas. Sendo assim ele mantém uma "compatibilidade conceitual" com os padrões do Delphi anteriores ao DBX, comparáveis com o IBX, IBO, Unidac e BDE.

A performance do AnyDac chega bem perto do Unidac, tornando-se uma das melhores alternativas.

UIB

UIB ou Unified Interbase é a suíte de componentes da prodigy.com para acesso a Interbase e Firebird. Ela pode ser baixada a partir da url na seção de links e deve ser compilada e ter seus diretórios de sources adicionados na library path e searching path do Delphi assim como foi feito com a suíte Zeos, uma vez que o UIB não tem instalador.

De posse dos fontes descompactados existem dois arquivos de grupos de projetos, um para os pacotes de design time e outros para os pacotes de runtime.

O UIB é um substituto ou alternativo para o IBX e IBO que foi o preferido em uma época em que o IBX e IBO, e até mesmo o DBX não tinham suporte a todas as versões do Interbase. Na época da migração do Interbase 6.1/6.5 para o 7, algumas suítes de componentes mantinham compatibilidade com as versões antigas do Interbase mas não davam suporte a alguns recursos das versões novas. Outras faziam o contrário. Com o surgimento e amadurecimento do Firebird (que é um fork do Interbase, da época em que ele era open-source) o Firebird foi evoluindo à sua maneira, se distanciando e tornando-se bem diferente do Interbase. Alguns componentes como DBX e IBX perderam compatibilidade com o Firebird. Outros, como o IBO, fizeram o máximo esforço para manter essa compatibilidade.

Nesse cenário confuso surgiu o UIB, sigla para Unified Interbase, que tinha como proposta manter a compatibilidade tanto com Interbase quanto com Firebird, inclusive recursos especiais. E antes da Embarcadero desenvolver o tão esperado e desejado driver DBX para Firebird, a prodigy já tinha um driver DBX alternativo para Firebird. Sendo assim ele é muito semelhante ao IBX e IBO, tendo componentes de database, transaction e query, não sendo necessário, portanto, descrever como é feita a conexão, pois é redundante.

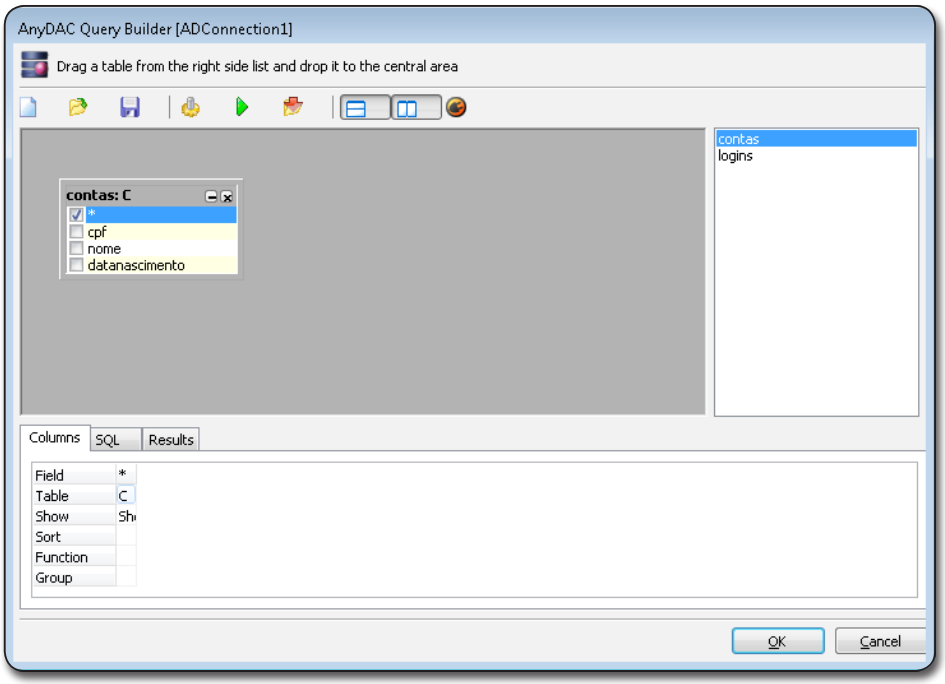


Figura 17. Query Builder

Como o Unidac, AnyDac e IBO são pagos, a maior vantagem do UIB, para quem vai usar apenas Interbase ou Firebird é que o UIB é open-source, licença MPL (Mozilla Public License), ou seja, o mesmo tipo de licença do browser Firefox.

Além de open-source, para acesso exclusivo ao Firebird ele se mostra mais rápido que DBX, IBX, IBO, AnyDac e Zeos, e um pouco mais rápido, nestes testes, que o Unidac, quase empatando. É uma ótima opção de conexão ao IB ou FB e uma alternativa barata e rápida ao DBX e outras suítes. O código fonte desta edição tem um exemplo de conexão com UIB.

Conclusão

Concluindo a leitura desta série de artigos o leitor estará apto a se conectar a qualquer banco de dados independente de qual seja, mesmo que ele não tenha sido apresentado aqui, pois já absorveu os principais conceitos.

Procure sempre por um driver DBX nativo, ou por um componente nativo, mesmo que seja específico de um único banco de dados. Componentes nativos oferecem a melhor performance e recursos extras, como componentes para monitorar SQL ou para gerenciar backups.

Com "componentes nativos" não queremos dizer nativos do Delphi, mas nativos (ou específicos) de uma determinada plataforma. Por exemplo, se sua aplicação for se conectar ao Oracle, procure por componentes específicos para Oracle, mesmo que tenha que comprá-los à parte. Geralmente o investimento é compensado em estabilidade e performance.

Prefira componentes que venham com código fonte, desenvolvidos por empresas tradicionais no mercado e confiáveis. Caso não encontre componentes específicos para determinado banco de dados, é praticamente impossível que um banco de dados hoje não tenha drivers OLEDB ou ODBC, então sempre é possível se conectar a eles de alguma forma.

Componentes open-source são sempre uma ótima alternativa, pois você tem

o direito de alterar o source, então são mais flexíveis do que os proprietários, mesmo que estes venham com fontes.

O novíssimo Delphi Starter Edition, a versão mais barata e mais enxuta do sistema, não vem com o DBX. Isso não é nenhuma limitação, pois podemos ver aqui que mesmo com um baixo orçamento, e mesmo sem componentes como Unidac ou AnyDac, ou mesmo IBO, ainda é possível se conectar com relativa performance a qualquer banco de dados utilizando o Zeos, e utilizando ADO ou outros componentes nos casos em que o Zeos não atenda às necessidades. A suíte de componentes Zeos permite criar projetos multibanco com a mesma flexibilidade que o DBX, ou migrar de banco, se for o caso, sem alterar nenhum componente, alterando apenas o "protocol". Se o seu caso for um projeto que acesse exclusivamente o Interbase ou Firebird, mesmo sem DBX, é possível se conectar ao banco de dados Firebird ou Interbase com significativa performance usando o UIB.

É possível tornar o sistema completamente independente de banco de dados e de componentes de acesso por aplicar os padrões Abstract Factory e Factory Method, explicados nas edições 122 e 123. A Edição 124 mostra como fazer essa camada de acesso isolada da aplicação usando esses padrões de projeto em conjunto com ClientDataset. São apresentados dois bancos de dados e duas formas de acesso. Juntando o exemplo daquele artigo com os conhecimentos deste é possível desenvolver uma camada de acesso a dados com muitas opções de bancos e de tecnologias de acesso, para vários cenários diferentes.

Além disso, é possível ainda distribuir aplicações desktop "standalone" que não necessitem de um servidor de banco de dados mas que salvem seus dados em XML ou com o SQLLite.

Os exemplos aqui mostrados também podem, sempre, ser separados em três camadas com a ajuda do DatasetProvider, ClientDataset e da tecnologia DataSnap. O código fonte fornecido juntamente com este artigo traz um Project group com todos os exemplos feitos neste mini-curso, mostrando exemplos de conexão com Oracle, DB2, MSSQL Server Express, Firebird, MySql, PostgreSQL, SQLite e MaxDB.

Vitor Luiz Rubio
vitorrubio@gmail.com - http://blog.vitorrubio.com.br
Tecnólogo em Processamento de Dados pela FATEC-SP e MCP. Atua como Analista de Sistemas Sênior na Editora Revista dos Tribunais. Programa com Delphi desde a versão 3 e com .Net desde a versão 1.1.

SAP DB 7.4
http://www.sapdb.org/7.4/sap_db_software.htm

Max DB 7.8
http://www.sdn.sap.com/irj/scn/maxdb-downloads

SQLite
http://www.sqlite.org/

Driver DBX para SQLite
http://www.devart.com/dbx/sqlite/download.html

Zeos
http://zeos.firmos.at/portal.php

Driver ODBC para DBExpress
http://sourceforge.net/projects/open-dbexpress/

Unidac
http://www.devart.com/unidac/download.html

IBO
http://www.ibobjects.com/

UIB
http://www.progdigy.com/?page_id=5

AnyDac
http://www.da-soft.com/download/anydac/anydac-trial/download.html

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Utilizando diagramas UML no Delphi XE

Fabrício Hissao Kawata

No âmbito do desenvolvimento de sistemas, o termo "modelo" pode ser entendido como uma abstração do mundo real. Para efetivamente se modelar um sistema, é necessário principalmente abstrair todos os seus detalhes irrelevantes que possam gerar algum tipo de entrave ou confusão. Outro quesito essencial é a utilização de uma linguagem com a qual o modelo possa ser descrito e lido pela máquina, podendo ser interpretado e transformado entre os sistemas. Uma linguagem de modelagem não necessariamente precisa ser feita de código real, podendo ser também um pseudocódigo, diagramas ou qualquer outra forma de descrever o sistema em si. Em sua composição, a linguagem de modelagem possui elementos que, juntos, formam sua notação, que expressa graficamente o modelo. Em vista disso, também se faz presente um termo chamado semântica (ou meta-modelo), que nada mais é que o conjunto de descrições relacionadas à notação. É a informação utilizada para descrever o significado de cada item da notação.

Modelagem com código de programação

Apesar de não viável, o próprio código de programação poderia ser um exemplo de linguagem de modelagem onde, no entanto, nenhum detalhe foi abstraído. Em outras palavras, toda a modelagem realizada seria tão somente sobre a implementação do sistema, ignorando outros pontos do mesmo, como contextualizar como e por quem o software será utilizado, além de exibir a execução de cada processo existente no sistema. Outro ponto que inviabilizaria utilizar o código de programação como linguagem de modelagem é quanto sua legibilidade. É fato de que código de programação não é legível para qualquer pessoa. Por exemplo, qual significado teria o trecho de código apresentado na **Listagem 1**, a uma pessoa que não fosse da área de desenvolvimento.

Dessa forma, seria necessário o emprego de outras técnicas para que se pudesse explicar e discutir o sistema com as outras partes interessadas do negócio, porém leigas, como é o caso dos usuários. Para esse tipo de pessoa pouco importa os detalhes do código fonte, a concentração deles se faz sobre os requisitos do sistema e como seus componentes irão trabalhar em conjunto para sanar suas necessidades de negócio. Além disso, para a própria equipe de desenvolvimento seria extremamente dificultosa a compreensão de partes ou do sistema como um todo somente através de código de software.

Resumo DevMan

De que trata o artigo:

Este artigo tem por finalidade apresentar um overview dos recursos oferecidos pelo IDE do Delphi XE para a modelagem de projetos UML, mais especificamente de seus diagramas. A partir de uma visão geral, o desenvolvedor Delphi se sentirá confortável em usufruir de tais recursos, por exemplo, em projetos orientados a objetos e outras situações cabíveis à construção de diagramas UML. Dessa forma, não será necessária a busca e consequente utilização de ferramentas de terceiros para a construção destes.

Em que situação o tema é útil:

Em termos teóricos, a UML (Unified Modeling Language) apresenta-se como uma linguagem visual para a modelagem de sistemas computacionais por meio do paradigma da Orientação a Objetos, tendo como objetivo documentar, estruturar e especificar melhor o seu desenvolvimento. Por estes motivos, ao passar dos anos acabou se tornando uma linguagem padrão de modelagem de software, sendo adotada mundialmente pela indústria de Engenharia de Software. O suporte à modelagem UML oferecido pelo IDE do Delphi XE, apesar de não se tratar de uma recente novidade, ainda é muito desconhecido pelos desenvolvedores da comunidade. De pronto, isto pode ser justificado pela excelência do Delphi em termos do desenvolvimento procedural, sobrepujando o desenvolvimento orientado a objetos. No contexto procedural não se tem a presença de um modelo de classes, que poderia justificar o uso da modelagem UML nestes projetos. Entretanto, cada vez mais se percebe o interesse desta comunidade no desenvolvimento OO, boas práticas, padrões de projetos, entre outros temas que visam uma melhoria na forma de programar. Sendo assim, gradativamente itens do mundo OO serão de interesse destes programadores, e com certeza os diagramas UML serão de suma importância neste determinado momento.

Diagramas UML:

O foco do artigo é a construção de diagramas UML a partir do próprio IDE do Delphi XE, visto que o desconhecimento sobre o assunto poderia gerar a procura por outras ferramentas para o mesmo propósito. Além disso, os desenvolvedores que agora estão fazendo uso dos conceitos da Orientação a Objetos em seus projetos Delphi, certamente farão uso dos recursos aqui apresentados. É importante ressaltar que o artigo não tem pretensões de especificar a UML, e sim fazer um apanhado geral de seus conceitos, ao mesmo tempo em que mostra os recursos disponíveis no IDE. Portanto é de suma importância o conhecimento prévio do assunto para o bom aproveitamento deste artigo.

Listagem 1. Trecho de código de uma classe de um sistema

```
TPessoa = class
 private
  FId: Integer;
  FNome: string;
  procedure SetId(const Value: Integer);
  procedure SetNome(const Value: string);
  { private declarations }
 protected
  { protected declarations }
 public
  { public declarations }
 published
  { published declarations }
  property Id: Integer read FId write SetId;
  property Nome: string read FNome write SetNome;
 end;
```

Modelagem com linguagem informal

Linguagem informal é aquela que não possui uma notação definida (formal), ou seja, é tão flexível que não apresenta regras bem delineadas, dificultando sua clareza, o que vai de encontro do que se espera de uma linguagem de modelagem ideal, conforme será visto adiante. Um exemplo clássico de linguagem informal é a nossa própria linguagem natural, que é a utilizada, por exemplo, para escrever o presente artigo. Ou ainda a linguagem que utilizamos em nossas conversas do dia a dia. No entanto, quando se trata de modelagem de sistemas toda esta flexibilidade da linguagem informal se torna um problema. Devido a sua abertura, ela tende a apresentar um grau elevado de ambiguidade em seu emprego, podendo um simples texto ser interpretado de formas diferentes, por pessoas diferentes. Como exemplo do emprego da linguagem natural, poderíamos descrever o que foi apresentado na **Listagem 1** da seguinte forma: "TPessoa é uma classe do sistema que possui seis membros, dos quais dois são métodos privados, dois são propriedades públicas e outros dois são fields privados...". Consequentemente, a partir desta forma escrita, dependendo do entendimento de quem o está interpretando, poderia se obter uma notação gráfica igualmente tão imprecisa e ambígua. Analisando ainda outro aspecto, por não apresentar uma precisão satisfatória, a linguagem informal não poderia também ser futuramente transformada em código de programação, pois se corre o risco de ter códigos diferentes sendo gerados a partir de uma mesma fonte, justamente por sua ambiguidade.

Modelagem com linguagem formal

Conforme visto, em termos de modelagem as duas formas de linguagem apresentadas até aqui não se mostraram satisfatórias como solução. Enquanto que a linguagem de código de programação mostra-se demasiadamente detalhada para o contexto, a linguagem informal é na mesma intensidade ambígua. Do lado oposto destas temos a linguagem formal, que se estabelece como a linguagem de modelagem ideal. Diz-se formal devido à linguagem possuir uma notação simples o suficiente para ser facilmente aprendida, apresentando regras e significados bem definidos. Assim sendo, diante das várias partes envolvidas no projeto, a partir do conhecimento da linguagem formal, o projeto seria mais claramente entendido por todos os envolvidos.

É neste ponto que se apresenta a UML, que por todas estas características se mostrou ao longo dos anos muito bem estabelecida no meio do desenvolvimento. Na **Figura 1** temos um exemplo simples de notação UML para a classe anteriormente descrita.

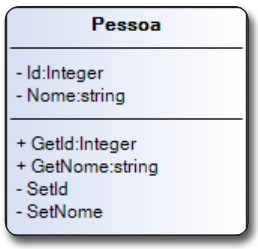


Figura 1. Notação UML para a declaração de uma classe

UML

UML, termo acrônimo para Unified Modeling Language, é uma linguagem de modelagem padrão no que tange o desenvolvimento de softwares. Uma modelagem UML permite documentar os aspectos importantes do projeto, de forma a gerenciar a complexidade do mesmo. Ela visa, entre outras coisas, manter o controle dos componentes necessários, bem como a empregabilidade de cada um no contexto do sistema que está sendo desenvolvido para que, em conjunto, eles cumpram o papel principal que é o de atender os requisitos do cliente. A UML é também popularmente conhecida como o resultado das melhores práticas da comunidade orientada a objetos durante as últimas décadas e mostra-se em constante evolução, conforme novas necessidades vão surgindo. A seguir são apresentados alguns dos pontos onde podemos justificar a escolha para a utilização da UML:

- Concisão: a base da UML são suas notações, que por sua vez são concisas, ou seja, simples e diretas. Isso facilita o aprendizado da linguagem;
- Abrangência: com a UML pode-se, de forma fácil e efetiva, descrever todos os aspectos importantes de um sistema;
- Escalabilidade: a linguagem utilizada visa evitar os exageros, de tal forma a ser formal o suficiente para a modelagem de grandes projetos como também o inverso, para o caso de projetos menores;
- Padrão: a UML não possui vínculo com qualquer produto em específico. É controlada por um grupo de fornecedores e acadêmicos que garantem sua interoperabilidade, podendo atender a vários produtos distintos, tornando-se padrão entre estes;

Historicamente falando a UML surgiu da união das três metodologias de modelagem mais populares entre os profissionais da área da engenharia de software até a década de 90: o método Object Modeling Tecnique (OMT), o método Booch e o método Object-Oriented Software Engineering (OOSE). Logo em sua primeira versão, no ano de 1996, a UML realmente retirou a ambiguidade

na comunicação das pessoas envolvidas, utilizando um design que efetivamente transmitisse a essência do projeto de forma a mapear seus requisitos funcionais. Tão logo grandes empresas da área de engenharia e desenvolvimento de software passaram a contribuir com o projeto. Isto culminou com a adoção da UML com linguagem padrão de modelagem, já no ano seguinte, pela OMG (Grupo de Gerenciamento de Objetos).

No entanto, como todo começo, apresentou algumas falhas que foram corrigidas com o tempo. Dentre estas podemos citar sua incapacidade de compartilhar seus modelos entre ferramentas diferentes, uma vez que todas suas revisões até a versão 1.x tinham sido concebidas como uma linguagem unificada voltada para seres humanos. Descobriu-se então que seu meta-modelo não era formal o suficiente para a partilha entre máquinas. Já em sua versão 1.5 a UML atingia certo nível de maturidade, podendo descrever razoavelmente bem um sistema. Contudo, ainda apresentava certo grau excessivo de complexidade, dificultando sua extensão, quebrando seu objetivo fundamental de ser uma linguagem formal e simplista. Era então o momento ideal para mudanças, uma reestruturação que resultou em sua versão 2.0. Nesta versão novos tipos de diagramas foram introduzidos de forma a auxiliar as práticas mais recentes (Ex: diagrama de pacotes e diagrama de tempo) e sua notação foi ampliada, sempre com o cuidado de tomar como base o que já havia sido estabelecido nas versões anteriores.

Diagramas UML

Em UML os diagramas são formas especiais constituídas de uma coleção de elementos a fim de se mostrar partes de um modelo. Portanto, quando se manipula um diagrama na notação UML, está se manipulando uma visão do conteúdo do modelo. Dessa forma, nem tudo que está presente no modelo necessariamente precisa estar em um diagrama. Esta definição acaba com a falsa impressão inicial que se tem, quando se inicia em UML, de que o conjunto de diagramas construídos forma essencialmente o modelo do sistema.

Os diagramas ainda são divididos em duas categorias:

- Estruturais: referem-se aos aspectos estáticos de um sistema, ou melhor dizendo, sobre sua estrutura estável (seu esqueleto). Dessa forma, estes diagramas servem para que se possa visualizar, especificar e documentar tais aspectos do sistema. São exemplos de diagramas estruturais: diagrama de classe, diagrama de componentes, diagrama de pacotes, diagrama de objeto, diagrama de implantação e diagrama de estrutura;
- Comportamentais: ao contrário dos diagramas estruturais, os diagramas comportamentais referem-se aos aspectos dinâmicos de um sistema. São utilizados para que se possa visualizar, especificar e documentar as partes do sistema que sofrem alterações ao longo do tempo, como seus fluxos de mensagens e movimentação de componentes. São exemplos de diagramas comportamentais: diagrama de caso de uso (Use Case), diagrama de atividades e diagrama de transição de estados. Ainda nesta mesma categoria pode-se atribuir outra subcategoria denominada Diagramas de Interação. Estes diagramas, além das características comportamentais, exibem uma interação entre os elementos ou um conjunto de elementos do sistema e as mensagens trocadas por eles, estabelecendo assim uma visão ainda mais dinâmica do próprio sistema. Diagrama de sequência, diagrama de interatividade, diagrama de comunicação e diagrama de tempo, são alguns exemplos desta subcategoria.

Diagramas UML no IDE do Delphi

Como já mencionado, o próprio IDE do Delphi XE provê a construção de alguns dos principais diagramas UML. O ponto de partida para isto está presente em suas próprias opções de projeto, através de dois itens nomeados como UML Design Project, conforme pode ser visto na **Figura 2**.

Uma vez criado um projeto voltado para a UML 2.0 (que é a versão mais recente suportada pelo IDE), tem-se as seguintes opções de diagramas a serem construídos, visualizados na **Figura 3**.

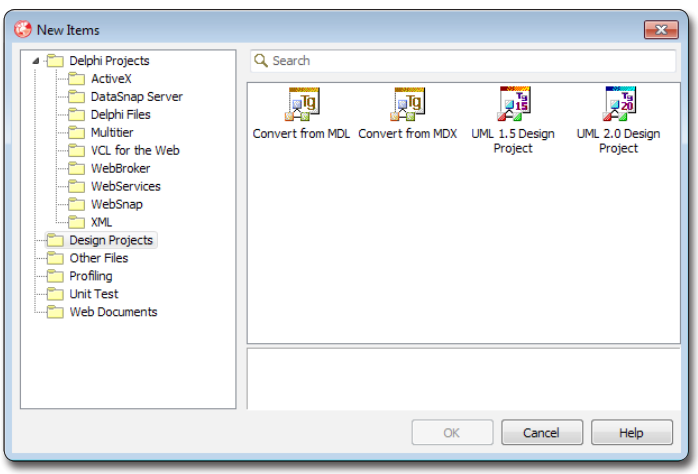


Figura 2. Opções de projetos UML

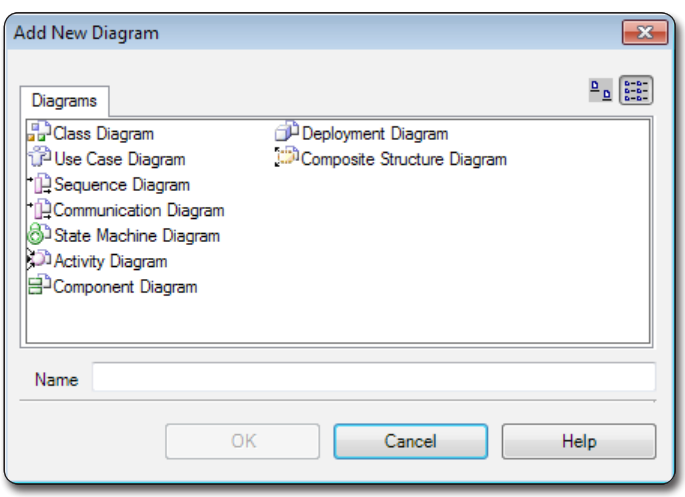


Figura 3. Diagramas UML disponibilizados pelo IDE

Diagrama de Classe (Class Diagram) – modelando a estrutura lógica do sistema

Uma Classe é o modelo para um objeto que poderá ser construído, ou seja, ela contém toda a especificação deste objeto, todas as suas características. Como exemplo, temos seus atributos e métodos. Um atributo é uma peça de informação que representa o estado do objeto. Já um método descreve a operação que uma classe poderá fazer. O conjunto dessas operações define o comportamento da classe. Dito isto, o Diagrama de Classes é a representação de todas as classes que o sistema deverá possuir. Justamente por este motivo pode-se afirmar que é um dos diagramas mais utilizados e fundamentais na UML, servindo de apoio inclusive, para a grande maioria dos demais diagramas. Neste diagrama são explicitados também os atributos e métodos de cada classe, além dos relacionamentos entre elas.

A definição dos relacionamentos entre as classes é outro fator imprescindível neste tipo de diagrama. Isto acontece, pois a forma de envolvimento de uma classe com outra pode ser de vários tipos:

• Dependência: este tipo de relacionamento pode ser considerado a mais fraca relação direta que pode existir entre duas classes. Isto pode ser explicado da seguinte forma: quando há uma relação de dependência entre duas classes de objetos, simplesmente garante-se que elas dependem uma da outra para trabalhar em tempo de execução;

• Associação: em suma uma associação indica que uma determinada classe contém uma referência a um objeto(s) de outra classe, na forma de atributo. A definição de origem e destino das informações representa a navegabilidade do relacionamento;

• Agregação: este é um tipo especial de associação, pode-se assim dizer. De forma clara, uma agregação é a representação de um todo composto de várias partes. Isso indica que uma classe contida no relacionamento pode ser uma parte ou estar contida em outra classe. Não há restrições, visto que, uma parte não depende essencialmente da outra para existir;

• Composição: também é considerado um tipo especial de associação, da mesma forma que a Agregação. O que o difere do anterior é seu caráter de contenção. Em síntese, neste tipo de relacionamento, classes de objetos que estão contidas dentro de outra classe e dependem desta para existir. Esta classe acaba funcionando como uma espécie de container para as outras. Desta forma, os elementos são criados e destruídos de acordo com seu container;

• Generalização (ou herança): este tipo representa o relacionamento entre um elemento mais geral e outro mais específico. Isso ocorre quando uma determinada classe é na verdade um tipo de outra classe, havendo assim uma relação de herança entre elas. Como exemplo, uma classe especializada além de herdar todos os atributos e métodos de seu ancestral (classe genérica), pode ainda ter a adição de outros atributos e métodos adicionais, específicos para os casos que venha a atender.

No Delphi, todos estes itens citados estão disponíveis na forma de componente, assim como todos os outros complementares, para que seja possível construir um Diagrama de Classes da melhor forma. Conforme o tipo de diagrama que está se construindo, a Tool Palette do IDE é personalizada, fornecendo os elementos condizentes com cada um. A **Figura 4** mostra em detalhe a paleta de componentes carregada especificamente para o Diagrama de Classes da UML 2.0.

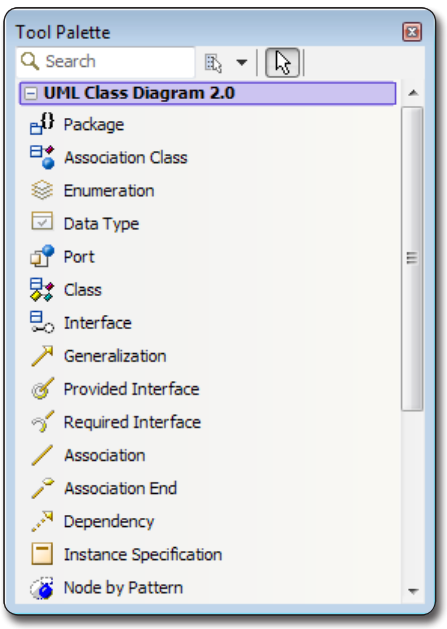


Figura 4. Tool Palette exclusiva para diagramas de classe

Diagrama de caso de uso (Use Case) – modelando requisitos

É através do Diagrama de Caso de Uso que se consegue uma visão ampla do sistema que está sendo projetado. Através de uma representação gráfica, ilustra os elementos que irão interagir com as funcionalidades do sistema, auxiliando também no levantamento de requisitos (que determinam as necessidades do usuário). Por este motivo serve de base para todos os demais diagramas. É considerado ainda o diagrama mais informal da UML, pois faz uso de uma linguagem extremamente simples e, portanto, de fácil compreensão até mesmo para uma pessoa sem o conhecimento técnico devido. Assim sendo, um usuário pode facilmente ter uma visão prévia de como o sistema irá se comportar.

Para a construção de um Diagrama de Caso de Uso, faz-se uso de dois itens essenciais para a representação do contexto: Ator e Caso de Uso. Um ator, neste caso, trata-se de uma entidade que irá interagir com o sistema, ou seja, o seu utilizador. Apesar de sua nomenclatura, ele não necessariamente representa uma pessoa real (ser humano), podendo ilustrar outro sistema ou mesmo uma máquina. Entretanto, detalhes do ator não são o foco, e sim a sua interação com o sistema. No diagrama o ator toma a forma de um boneco para sua representação. Já um caso de uso tem por finalidade descrever a maneira como o sistema se comporta para atender determinada exigência do usuário, para a execução de um trabalho específico. Essa descrição ainda determina a relação entre o ator e o próprio sistema. Em termos de representatividade, um caso de uso ganha a forma de elipse no diagrama.

Em um diagrama de caso de uso pode-se ainda representar as relações existentes entre os elementos envolvidos (atores e casos de uso). Basicamente são três as possibilidades. A primeira delas diz respeito à relação entre atores, que se dá num contexto de generalização (especialização), conforme conceito já apresentado anteriormente. Esta definição é utilizada para que seja demonstrada a sobreposição de papéis entre atores, ou seja, representar que determinado ator é um tipo especial de outro, inclusive com atribuições adicionais.

As relações entre casos de uso são nomeadas como include e extend. No include, um caso de uso "inclui" outro, ou seja, numa relação direta existente entre dois casos de uso (1 e 2) o comportamento de 1 é completamente inserido em 2. Na prática, o caso de uso 2 deverá reutilizar todas as etapas do caso de uso 1. Esta relação ainda estabelece um contrato de obrigatoriedade, onde a ocorrência do caso de uso 1 invariavelmente ocorre no momento do caso de uso 2. Já no extend, um caso de uso pode estender de outro. Sua principal diferença em relação ao anterior é que, neste caso, o comportamento de um caso de uso estendido não obrigatoriamente será reutilizado por outro caso de uso. A notação gráfica de ambos, include e extend, se dá por meio de setas com a prototipagem <<include>> e <<extend>>, respectivamente.

Novamente, no próprio IDE do Delphi estão presentes os recursos necessários para a construção deste tipo de diagrama. Na **Figura 5** temos a forma da paleta de componentes ao se trabalhar com um Digrama de Caso de Uso. Nela notamos a presença dos itens citados até aqui, como ator (Actor), caso de uso (Use Case), generalização (Generalization), extend (Extends) e include (Includes). Na **Figura 6** é apresentado um protótipo de um Diagrama de Caso de Uso utilizando-se tais componentes.

Diagrama de sequência – modelando a ordem das interações

O Diagrama de Sequência representa a ordem das interações entre as partes do sistema, de forma simples e lógica. Com isso, quando determinado caso de uso é executado é possível descrever cada interação acionada e a ordem em que elas ocorrerão. Quanto maior a granularidade de classes e consequentemente de seus métodos, aumenta-se a dificuldade em determinar a sequência de operações (comportamentos) global. Entre os diagramas de interação, o Diagrama de Sequência pode ser considerado o mais popular, uma vez que, até mesmo os novatos em UML conseguem achar algum sentido em suas representações.

Dentre os principais membros de um Diagrama de Sequência podemos citar a linha de tempo (de vida). Como neste tipo de diagrama são descritas as ordens em que as interações acontecem, o tempo se torna um fator fundamental. Todavia, o tempo neste caso faz referência à ordenação e não sobre a duração de uma interação. O objeto é outro membro a ser destacado no conceito de um Diagrama de Sequência, uma vez que representa a instância de uma classe (objeto) no processo. Em uma interação mensagens são trocadas entre os objetos do sistema ou, melhor dizendo, um evento é passado de um objeto a outro objeto que funcionará como receptor. Assim sendo, as mensagens também são partes integrantes do diagrama. Por fim, outro membro a ser destacado são as barras de ativação (activation bars), que são mostradas nas extremidades de envio e recebimento de uma mensagem, indicando que, por exemplo, o objeto que está enviando a mensagem estará ocupado durante o processo. Da mesma forma o objeto que a recebe estará também ocupado após a mensagem recebida.

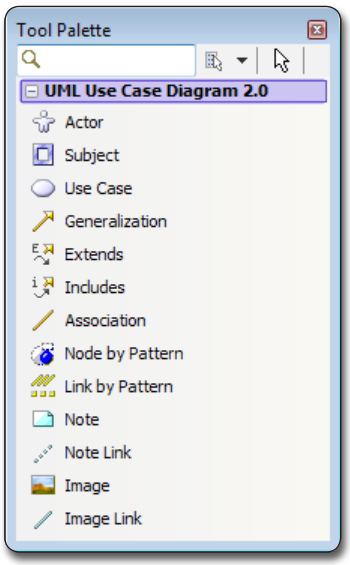


Figura 5. Tool Palette exclusiva para diagramas de caso de uso

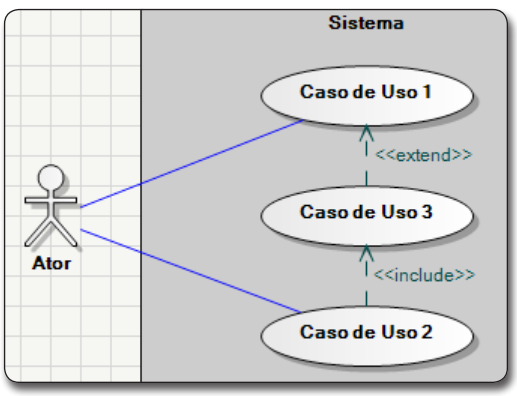


Figura 6. Protótipo de um Diagrama de Caso de Uso produzido no IDE

No IDE, a Tool Palette referente a um Diagrama de Sequência é exibida na **Figura 7**.

Diagrama de comunicação – modelando relações e interações

Até a versão 1.5 da UML, o Diagrama de Comunicação era então conhecido como Diagrama de Colaboração, no entanto com o surgimento da versão 2.0 sua nomenclatura foi modificada. Em termos conceituais, enquanto um Diagrama de Sequência refere-se à ordem das interações entre as partes do sistema, um Diagrama de Comunicação provê outra perspectiva destas interações, centrada estritamente na relação entre as partes. Desta forma consegue-se visualizar quais partes devem estar conectadas para que uma interação possa ocorrer. Neste tipo de diagrama a ordem dos eventos relacionados a uma interação possui uma importância secundária.

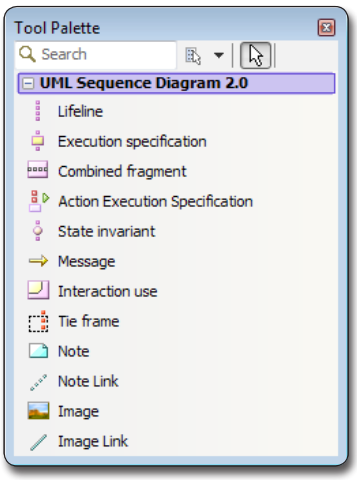


Figura 7. Tool Palette exclusiva para diagramas de sequência

O diagrama de comunicação possui três itens envolvidos em sua composição, conforme pode ser visto no protótipo construído no IDE e mostrado na **Figura 8**: participante (ou objeto), ligação e mensagem. O participante diz respeito ao elemento envolvido como, por exemplo, a instância de uma classe. A ligação (em forma de linha) representa o vínculo entre os participantes e tem como finalidade mostrar que mensagens podem ser trocadas nesta relação que se estabelece. Quando não há uma ligação, não há também uma interação entre os participantes. Já a mensagem está relacionada a um método do sistema.

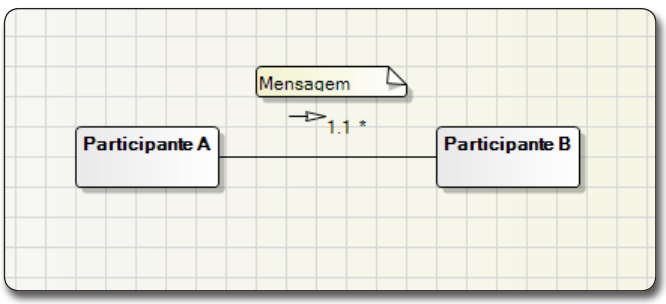


Figura 8. Protótipo de Diagrama de Comunicação

Diagrama de atividade – modelando fluxos de trabalho

Um Diagrama de Atividade, essencialmente, descreve os passos que o sistema deve percorrer para que uma determinada atividade específica seja concluída. Ao contrário de um caso de uso que especifica o que o sistema deve fazer, um Diagrama de Atividade especifica de que forma (como) o sistema irá atender aos seus requisitos. Para isso, de forma gráfica, são mostrados os fluxos das atividades envolvidas. Conclui-se então que este tipo de diagrama se torna uma espécie de fluxograma, ajudando inclusive na modelagem de negócio.

Os conceitos envolvidos neste tipo de diagrama são de grande quantidade, no entanto vale destacar três deles: atividade, ação e decisão. O termo atividade é utilizado para representar o processo que está sendo modelado. Aproveitando este conceito, é possível definir uma ação, que é a simbolização de uma medida a ser tomada para a conclusão da atividade. Enquanto isso, uma decisão indica que mediante uma condição estabelecida, diferentes ações poderão ser executadas.

Assim como mostrado em alguns dos exemplos anteriores, todos os elementos necessários para a construção de um Diagrama de Atividade são disponibilizados pelo IDE (**Figura 9**).

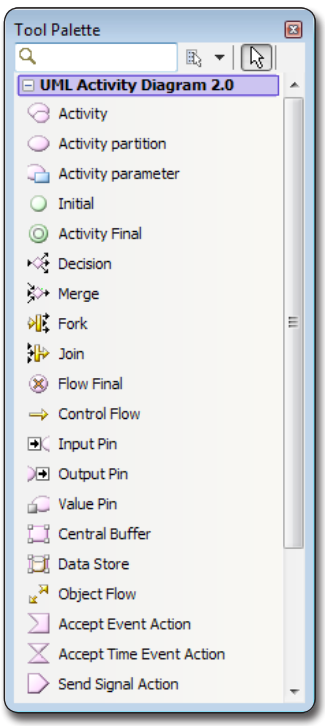


Figura 9. Tool Palette para Diagramas de Atividade

Diagrama de máquina de estados – modelando estados de um objeto

O Diagrama de Máquina de Estados, em versões anteriores a 2.0 da UML, era conhecido como Diagrama de Estados. Ele demonstra graficamente o comportamento de um elemento mediante a ocorrência de suas transições de estado. Uma transição de estado representa uma mudança. Já o elemento é considerado um objeto, podendo ser, além de uma instância de classe, um caso de uso ou até mesmo um sistema. Isto acontece porque um Diagrama de Máquina de Estados pode também ser usado para modelar o comportamento de um caso de uso ou de um sistema por completo. Por estado, entenda como sendo a situação em que um objeto se encontra em dado momento do processo em que participa. Uma ilustração dos recursos oferecidos pelo IDE para a construção de Diagramas de Máquina de Estados é mostrado na **Figura 10**. Esta imagem foi extraída do projeto de demonstração, denominado UML 2.0, que acompanha a ferramenta.

Diagrama de componentes – modelando as peças do sistema

Durante o estabelecimento da arquitetura de um sistema, de certa forma está se planejando as partes que farão parte de sua

composição, mensurando a complexidade e dependência entre elas. É neste ponto que surgem os componentes que têm a finalidade de organizar o sistema em partes gerenciáveis, reutilizáveis e substituíveis. Isso quer dizer que eles deverão poder ser alterados sem que mudanças forçadas ocorram em outras partes do sistema. Um componente pode ainda estar relacionado a interfaces, que é um modo de consumir e fornecer comportamento numa interação com outros componentes.

Assim sendo, o Diagrama de Componentes mostra os componentes que fazem parte da composição de um todo. Este "todo" pode ser um sistema, um próprio componente ou ainda outro elemento do sistema. Na **Figura 11** são apresentados dois dos elementos disponíveis para a construção de Diagramas de Componentes na Tool Pallete do IDE.

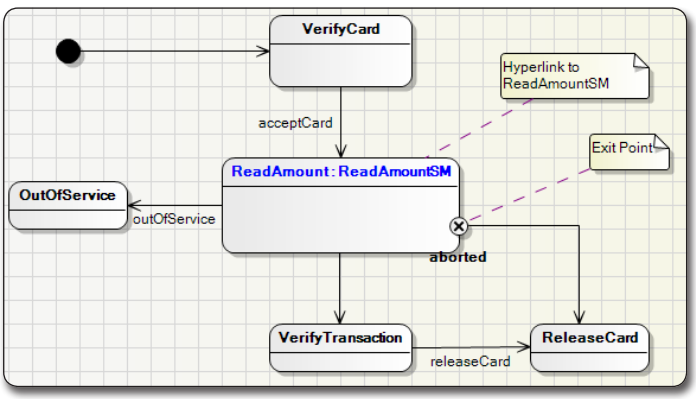


Figura 10. Trecho de um Diagrama de Máquina de Estados de um projeto real

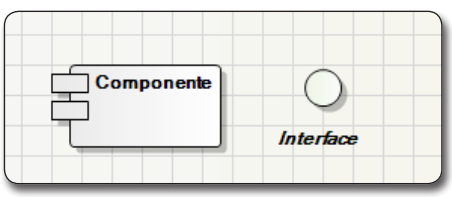


Figura 11. Representação de um componente e uma interface, a partir do IDE

Diagrama de implantação (Deployment Diagram) – modelando as necessidades físicas do sistema

O Diagrama de Implantação representa visualmente as necessidades do sistema do ponto de vista físico. Desta forma o sistema é trazido para um âmbito mais real, indicando todo o aparato de hardware sobre qual o sistema deverá ser executado, como estações de trabalho, servidores, topologias de rede, etc.

É provável que se aponte com exatidão as reais necessidades do sistema somente ao final do projeto, no entanto, um diagrama de implantação mostra-se útil em todas as fases do processo. Por exemplo, ao se iniciar o desenho de um novo projeto, somente as informações básicas da arquitetura física são conhecidas. Já neste momento é possível modelar essas características definindo um Diagrama de Implantação inicial. Na **Figura 12** é apresentado um fragmento de um Diagrama de Implantação fictício, que exibe dois dos principais elementos envolvidos neste tipo de diagrama: artefato e nó. Os artefatos são os arquivos físicos utilizados pelo sistema tais como arquivos executáveis (como o apresentado na imagem), arquivos DLL, arquivos de configuração, entre outros. Já o nó é o local onde o artefato está sendo implantado, ou seja, onde ele irá residir. No exemplo mostrado, o artefato reside em um PC Desktop.

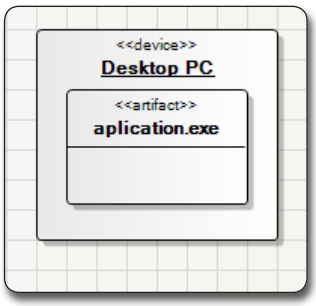


Figura 12. Fragmento de um Diagrama de Implantação

Diagrama de estrutura composta – modelando a colaboração de elementos

O Diagrama de Estrutura Composta é similar ao Diagrama de Classes, todavia, enquanto um Diagrama de Classes representa uma estrutura mais geral das classes do sistema, um Diagrama de Estrutura Composta modela esta estrutura de forma mais específica. Além disso, um Diagrama de Estrutura Composta é essencialmente utilizado para modelar colaborações. Uma colaboração pode ser entendida como a visão de um conjunto de classes (representadas por suas instâncias), que cooperam entre si a fim de atingir um objetivo em comum como, por exemplo, a execução de uma função específica.

Na **Figura 13** temos o esboço de um Diagrama de Estrutura Composta construído no IDE do Delphi, utilizando os recursos proporcionados por ele. Nesta imagem é apresentada uma colaboração denominada Aula, onde há o envolvimento de dois elementos, Aluno e Professor, que apesar de serem do mesmo tipo (Pessoa), apresentam papéis diferentes no contexto.

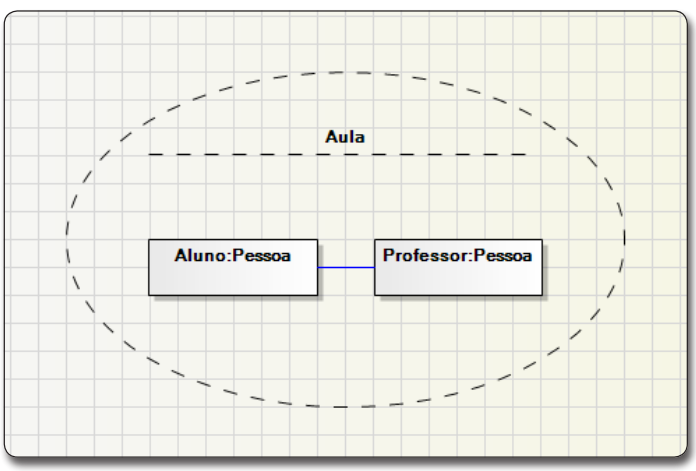


Figura 13. Diagrama de estrutura composta feito no IDE

Conclusão

Como já mencionado, no desenvolvimento de sistemas orientado a objetos, a presença de uma linguagem de modelagem eficaz é de suma importância, já que é através de sua notação que podemos expressar visualmente o sistema de uma forma geral. Neste momento a UML mostra-se então como uma linguagem padrão muito bem estabelecida. Como não poderia deixar de ser, o Delphi apresenta em seu IDE recursos para o uso da UML em seus projetos. Um exemplo disso pôde ser comprovado neste artigo, onde foi apresentada uma visão geral sobre o seu suporte à construção dos diferentes tipos de diagramas da UML.

Fica como dica, ou até mesmo desafio, a construção destes diagramas na mesma intensidade em que se dá construção de seus projetos OO em Delphi. Aposte no IDE.

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ. Atua como Analista Programador Delphi há 5 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas além de participar do programa Beta Tester Delphi Prism, da Embarcadero.

RAD Studio XE - Página de download
https://downloads.embarcadero.com/free/rad_studio

RAD Studio XE - Página oficial do produto
http://www.embarcadero.com/products/rad-studio

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Inversão de Controle e Injeção de Dependência

Rafael Stavarengo

Diz-se que um projeto está altamente acoplado quando suas classes e módulos conhecem muito um do outro, criando uma grande dependência entre eles. Isto acontece, na maioria das vezes, devido a uma decisão errada, tomada pelo projetista, que aos pouco encaminha o projeto para um design caótico onde uma simples manutenção de rotina pode desencadear uma série de erros em outro ponto do projeto.

O grande vilão, que pode tornar uma alteração pequena em um verdadeiro desastre é o relacionamento entre as classes, por isso é vantajoso investir o tempo necessário (ou possível), para criar designs onde as classes sejam capazes de se relacionar conhecendo o mínimo uma da outra. Criar classes coerentes que dependem apenas de interfaces é uma das formas de garantir o baixo acoplamento, porém o princípio "programe para interface", sozinho não ajudará muito.

O problema do alto acoplamento

Podemos medir o quão fortemente as classes de uma aplicação estão conectadas, possuem conhecimento ou dependem de outra classe observando o acoplamento existente em cada caso. Quanto mais acoplada uma classe estiver de outra maior o risco de termos uma combinação dos seguintes problemas:

- Mudanças numa classe relacionada forçam alterações locais à classe para que o funcionamento seja mantido;
- O reuso da classe torna-se mais difícil uma vez que sua implementação depende de outras classes;
- A coesão da classe diminui na medida em que aumenta o acoplamento caso ela assuma parte das responsabilidades de outra classe;
- A classe não pode ser testada isoladamente já que depende de testes em conjunto com outras classes, aumentando inclusive o esforço de entendimento.

Percebemos rapidamente que diminuindo ao máximo o acoplamento além de evitarmos os problemas mencionados anteriormente tornaremos nossa aplicação mais flexível e fácil de ser mantida, especialmente em equipes com paralelismo de desenvolvimento de atividades correlatas.

Utilizando as corretas técnicas e ferramentas para injeção de dependência, diversos componentes podem ser desenvolvidos paralelamente (aproveitando de recursos como Mock Objects para auxiliar nos testes unitários) e sua aplicação se tornará mais plugável e flexível, exigindo um esforço menor para modificações.

Resumo DevMan

De que trata o artigo:

Este artigo mostra o que é e como usar inversão de controle e a injeção de dependência. Você aprenderá na prática, através de um exemplo, como estes princípios de programação orientada a objetos podem ser usados e qual as vantagens que eles trazem.

Em que situação o tema é útil:

A inversão de controle torna um projeto mais escalável e simples de entender. Este é um conhecimento obrigatório para qualquer programador que se preze. Este artigo ajudará você a entender porque a herança deve ser evitada explicando onde a inversão de controle pode ser usada para substituí-la.

Inversão de Controle e Injeção de Dependência:

Como desenvolvedores de software, estamos sempre preocupados em criar um projeto escalável, e nos esforçamos para isto. Para atingir este objetivo é possível usar padrões, técnicas e até filosofias de desenvolvimento diferentes. Indiferente de como sua equipe escolheu trabalhar, todo projeto flexível tende para o mesmo lado: diminuir o acoplamento entre as classes. Este artigo explica como usar a inversão de controle para diminuir o acoplamento e apresenta algumas opções para injetar automaticamente as dependências de uma classe a fim de conseguir um projeto mais flexível.

Técnicas para resolver o problema de alto acoplamento

Tomemos como exemplo uma classe chamada ClasseA que depende dos serviços Servico1 e Servico2, conforme a **Figura 1**.

Se em algum momento necessitarmos substituir ou atualizar alguma das suas dependências, a classe ClasseA sofrerá alterações em seu código, uma vez que está relacionada diretamente com as classes de serviço. Outro ponto é que as implementações concretas das dependências precisam estar disponíveis no momento da compilação da solução. Uma das soluções seria delegar a função de selecionar a implementação da classe concreta para um componente (ou um container) externo.

Em diversas literaturas podemos encontrar esta solução sendo descrita como Padrão de Inversão de Controle (Inversion of Control Pattern, ou simplesmente IoC), onde a Injeção de Dependência é uma das maneiras de a implementarmos (assim como o Service Locator Pattern), conforme observamos na **Figura 2**. Sendo assim, é correto afirmarmos que a DI é um caso particular de IoC.

A classe ClasseA ao invés de fazer referência à classe concreta passa a referenciar uma interface IServicoA que por sua vez é implementada pelo serviço Servico1, o objeto identificado como Builder fica encarregado de resolver a classe concreta e injetar a dependência na classe ClasseA.

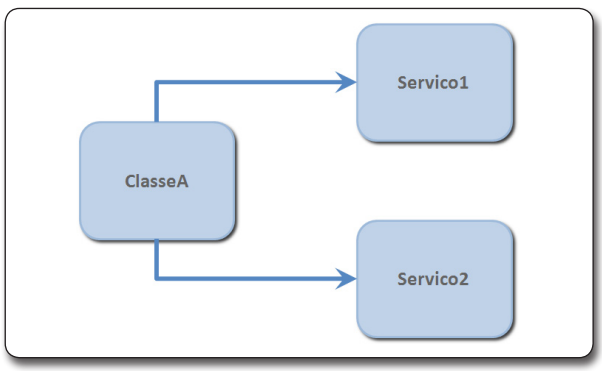


Figura 1. Dependência entre classes

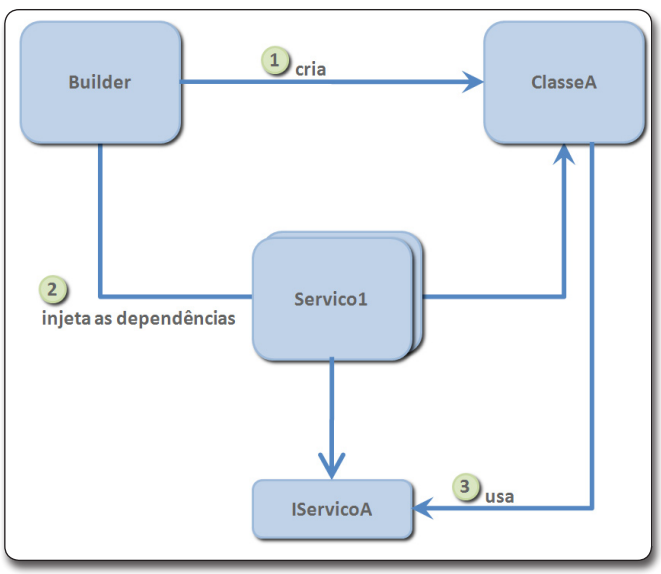


Figura 2. O padrão de Injeção de Dependência

Também é importante e necessário, que o projetista saiba como usar uma interface corretamente e para isto ele pode recorrer aos padrões de projetos. A maioria dos padrões de projetos usa interfaces e definem classes coerentes com baixa coesão para solucionar os problemas que se propõem a resolver. Mesmo assim, existem decisões, cujas respostas o projetista não vai encontrar em um padrão de projeto. Por exemplo: é melhor criar uma interface usando uma classe abstrata ou uma interface de fato? A resposta não é tão obvia quanto parece.

Outra forma de diminuir o acoplamento é a inversão de controle ou em inglês Inversion of Control (IoC). Inversão de controle é um princípio de orientação de objetos que define várias formas de simplificar um projeto invertendo o controle natural dos componentes. Existe mais de uma forma de inversão de controle: umas delas é a injeção de dependência.

A injeção de dependência ou em inglês dependency injection (DI), significa que uma classe não deve controlar as suas dependências, evidentemente tornando o projeto mais flexível. Nesta abordagem uma classe recebe suas dependências sem conhecer como elas são criadas ou de onde elas vêm. Pode parecer estranho para quem está escutando isto pela primeira vez, mas é isto mesmo que você leu. De forma superficial a injeção de dependência pode ser explicada assim: se a classe A precisa da classe B, então A não deve executar B.Create().

Como dito, existem outras formas de inversão de controle além da injeção de dependência. Outro exemplo disto está na própria VCL. Quando você cria um formulário e escreve qualquer instrução no evento OnCreate do mesmo, você está usando a inversão de controle, pois não é você quem controla a execução das instruções que você escreveu no evento, mas sim o framework, que neste caso é a VCL. É o framework que decide quando executar o código, e não o contrário.

Tipos de relacionamentos e o que eles têm a ver com DI

A técnica de injeção de dependência atua diretamente na forma como as classes e objetos se relacionam dentro de um projeto, por isto este tópico é dedicado a explicar quais sãos os tipos de relacionamentos existentes e a diferença entre eles. Você pode pular para o próximo tópico se estiver familiarizado aos jargões composição/agregação e já sabe por que dar preferência a eles é (quase sempre) melhor do que criar um relacionamento do tipo herança.

Quando a classe A usa a classe B, dizemos que as classes A e B estão relacionadas, seja usando a composição, agregação ou herança. Embora estes três tipos sejam igualmente comuns, a herança parece ser a mais difundida, talvez por ser a mais simples de compreender. Todos os três são simples de codificar, mas o conceito de herança é mais simples do que o conceito de composição e agregação. Mas porque devemos dar preferência à composição ou agregação ao invés da herança?

Herança é o que permite a uma classe herdar as características de outra. É como na herança genética, onde o filho herda características do pai e da mãe. Quando dizemos que TPatoDeBorracha é filha, ou subclasse de TPato, significa que TPatoDeBorracha herdou todas as características de TPato. Um problema neste conceito, é que se TPato tiver um bug, então TPatoDeBorracha e todas as demais subclasses de TPato, também terão o mesmo bug. Além disto, a herança é um tipo de relacionamento rígido que força as subclasses terem métodos mesmos que elas não precisem deles.

Para entender melhor este conceito, vamos montar um programa para simular o comportamento de patos usando herança.

O Simulador de Patos tem apenas uma classe: TPato. Esta classe implementa os comportamentos comuns de um pato real, como comer e grasnar. É um design simples, como você pode ver na **Figura 3**. Com o passar do tempo é necessário adicionar mais um tipo de pato ao simulador: os patos de borracha. Estes patos serão representados pela classe TPatoDeBorracha e para evitar digitar muito código podemos usar a herança para aproveitar o código que já foi escrito na classe TPato. O resultado é uma nova classe que herda as da classe TPato, veja como fica na **Figura 4**.

Figura 3. A classe TPato

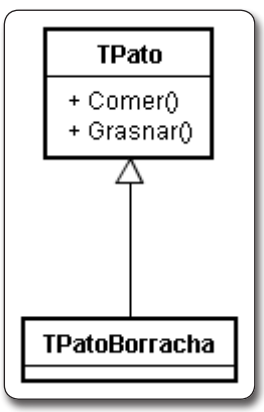


Figura 4. Criando um relacionamento com herança

Pense um pouco: um pato de borracha pode comer? Não faz sentido que a classe TPatoDeBorracha possua o método Comer(). Já o método Grasnar() tem algum sentido, pois os patos de borracha costumam fazer um barulho engraçado quando são apertados. Então, embora os patos de borracha possam grasnar, eles não grasnam da mesma forma que os patos reais. Estes problemas são simples de resolver graças à herança: basta sobrescrever os respectivos métodos da superclasse.

Agora temos uma classe TPatoDeBorracha que sobrescreveu todos os métodos da superclasse TPato. O método Comer() foi sobrescrito para anular o que era feito na classe TPato, já que os patos de borracha não podem comer. O método Grasnar() foi sobrescrito porque um pato de borracha emite um barulho diferente do grasnar de um pato real. Na **Listagem 1** está a classe TPato e na **Listagem 2** está a classe TPatoDeBorracha, que sobrescreve os métodos conforme dito anteriormente.

Listagem 1. Classe TPato

```
01  unit Pato;
02  interface
03  uses Dialogs;
04  type
05   TPato = class
06   public
07    procedure Comer;
08    procedure Grasnar;
09   end;
10    implementation
11    procedure TPato.Comer;
12    begin
13      ShowMessage('Um pato está comendo');
14    end;
15    procedure TPato.Grasnar;
16    begin
17      ShowMessage('Quack... Quack... Quack...');
18    end;
19  end.
```

Listagem 2. Classe TPatoDeBorracha

```
01  unit PatoDeBorracha;
02  interface
03  uses Pato, Dialogs;
04  type
05   TPatoDeBorracha = class(TPato)
06   public
07    procedure Comer; override;
08    procedure Grasnar; override;
09   end;
10    implementation
11   procedure TPatoDeBorracha.Comer;
12   begin
13     //Pato de borracha não come
14   end;
15   procedure TPatoDeBorracha.Grasnar;
16   begin
17     ShowMessage('Gueeeq... Gueeeq... Gueeeq...');
18   end;
19  end.
```

Chega a hora de criar mais um tipo de pato: TPatoEletronico. Os patos eletrônicos são mais parecidos com os patos reais do que os patos de borracha uma vez que eles não podem comer, mas podem grasnar. De fato, eles emitem o mesmo ruído que um pato real ao grasnar, portanto o método Grasnar() não precisa ser sobrescrito nesta nova classe. Já o método Comer() será reimplementado, pois patos eletrônicos não comem. Estas alterações encerram as modificações na hierarquia de patos, que criamos com a herança entre classes. Veja como ficou o design do projeto na **Figura 5**.

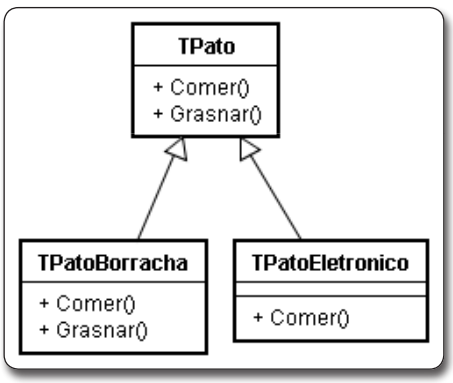


Figura 5. Um hierarquia criada com herança

Os pontos fracos da herança

Em um dado momento chega um novo requisito dizendo que os patos também precisam caminhar. Esta será uma tarefa simples. Basta aproveitar uma das vantagens da herança para criar um novo método na classe TPato para que os patos possuam o comportamento de "caminhar". Escolhi chamar este novo método de Caminhar(). Após esta simples alteração o Simulador de Patos é recompilado e a atualização liberada para os clientes. Pouco tempo depois chega a seguinte reclamação: "Os patos de borracha estão caminhando. Isto está certo?".

Claro que não, pois estes patos não possuem este comportamento e eles só estão "caminhando" porque nos esquecemos de sobrescrever o método Caminhar() em cada classe de pato que não caminha. De imediato é comum pensar que o erro foi esquecer-se de sobrescrever o método nas classes que não caminham, mas o erro vai além disto: o erro foi escolher a herança para este design.

Desde o princípio a herança vem gerando retrabalho para o Simulador de Patos: 1º) foi necessário reescrever os métodos da superclasse nas classes filhas; 2º) os métodos das superclasses não servem para todas as subclasses; 3º) implementar um novo método na superclasse, como foi o caso do método Caminhar(), pode ser totalmente desastroso.

O ideal para o Simulador de Patos é que os comportamentos sejam isolados das

classes de patos e assim cada pato possa escolher qual o comportamento que deseja para si próprio. Melhor ainda se eles puderem trocar este comportamento em tempo de execução. Isto é possível usando outro tipo de relacionamento entre as classes.

Um dos pontos fracos da herança é que ela fere o princípio "Aberto Fechado". Este princípio da programação orientada a objeto diz que as classes devem ser abertas para expansão, porém fechadas para modificações. Parece contraditório quando dito fora de contexto, mas de fato uma classe que não pode ser expandida sem precisar ser alterada pode gerar problemas graves, como aconteceu com Simulador de Patos.

Foi necessário fazer uma modificação na classe TPato para adicionar o comportamento "Caminhar" e devido ao uso da herança, modificar a classe TPato é o mesmo que modificar as classes TPatoDeBorracha e TPatoEletronico. Esta modificação gerou, sem que percebêssemos, um bug na classe TPatoDeBorracha. Para seguir o princípio Aberto Fechado é necessário encontrar outro design para o Simulador de Patos. Precisamos de um design que permita adicionar novos comportamentos na classe TPato, sem precisar alterar o código da classe.

É neste ponto que entra a composição/agregação juntamente com a injeção de dependência.

Criando um novo design

Para criar uma classe que respeite o princípio Aberto Fechado é necessário separar todo o código que vária em novas classes. No Simulador de Patos, alguns patos podem comer, enquanto outros não, por isso o método Comer() da classe TPato é um código que varia dependendo do tipo de pato que se pretende simular. A mesma regra vale para os métodos Grasnar() e Caminhar(): todos mudam de acordo com o tipo de pato que se deseja simular.

Uma vez identificado os pontos que mudam, devemos retirá-los da classe TPato e colocá-los em outras classes que, posteriormente serão injetadas, em tempo de execução, em um objeto de TPato. Para isto, foram criadas três novas classes: TComer, TGrasnar e TCaminhar. Cada uma destas classes é responsável pelo comportamento que seu nome descreve. Como você pode ver na **Figura 6**, as novas classes têm um método chamado "Executar", cujo objetivo é executar o comportamento que a classe se propõe.

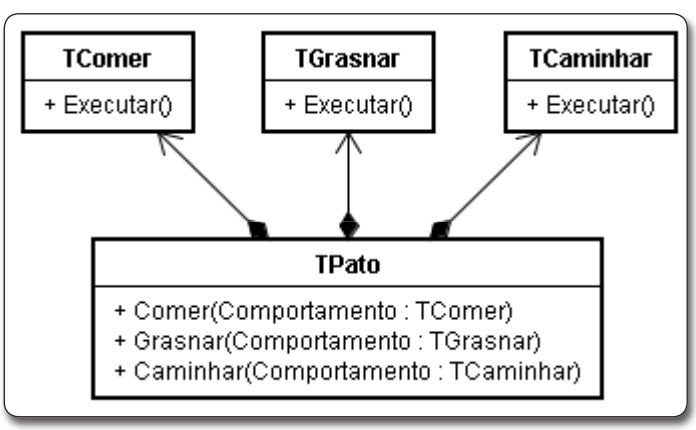


Figura 6. Relacionamento do tipo composição

Nota

O novo design do Simulador de Patos substituiu a herança pela composição, tornando o projeto mais fácil de entender. Agora é possível simular qualquer tipo de pato com apenas uma única classe de TPato. Quando uma classe A herda da classe B, dizemos que A é B, ou seja, TPatoDeBorracha é um TPato devido ao relacionamento do tipo herança. A composição ou agregação diz respeito ao fato de uma classe usar outra classe, ou seja, seria dito que a classe A tem uma classe B.

Ainda observando a **Figura 6**, é possível notar que os métodos Comer(), Grasnar() e Caminhar() da classe TPato passaram a receber um parâmetro. Isto foi feito porque, como já foi dito, o código dentro destes métodos foi removido e colocado no método Executar() das novas classes. Assim para que um pato possa, por exemplo, grasnar, ele precisa receber uma instância da classe TGrasnar, pois é esta classe que realmente sabe fazer um pato Grasnar. Veja como isto funciona: na **Listagem 3**, **Listagem 4** e **Listagem 5** estão as novas classes de comportamentos e na **Listagem 6** está a classe TPato alterada.

Listagem 3. A classe do comportamento "Comer"

```
01  unit Comer;
02  interface
03  uses Dialogs;
04  type
05   TComer = class
06   public
07    procedure Executar;
08   end;
09   implementation
10   procedure TComer.Executar;
11   begin
12     ShowMessage('Comendo...');
13   end;
14  end.
```

Listagem 4. A classe do comportamento "Grasnar"

```
01  unit Grasnar;
02  interface
03  uses Dialogs;
04  type
05   TGrasnar = class
06   public
07    procedure Executar;
08   end;
09   implementation
10   procedure TGrasnar.Executar;
11   begin
12     ShowMessage('Quack... Quack... Quack...');
13    end;
14  end.
```

Listagem 5. A classe do comportamento "Caminhar"

```
01  unit Caminhar;
02  interface
03  uses Dialogs;
04  type
05   TCaminhar = class
06   public
07    procedure Executar;
08   end;
09   implementation
10   procedure TCaminhar.Executar;
11   begin
12     ShowMessage('Caminhando...');
13    end;
14  end.
```

Agora, para simular um pato é necessário injetar os comportamentos dentro de uma instância de TPato. É algo como aparece na **Listagem 7**. Porém, depois destas alterações o Simulador consegue simular apenas patos reais já que os comportamentos criados com as classes TComer, TGrasnar e TCaminhar são comportamentos de patos reais. Ainda falta criar as classes para simular os comportamentos dos patos eletrônicos e dos patos de borracha.

Listagem 6. A nova classe TPato

```
01  unit Pato;
02  interface
03  uses Caminhar, Comer, Grasnar;
04  type
05   TPato = class
06   public
07    procedure Comer(Comportamento: TComer);
08    procedure Grasnar(Comportamento: TGrasnar);
09    procedure Caminhar(Comportamento: TCaminhar);
10   end;
11   implementation
12   procedure TPato.Caminhar(Comportamento: TCaminhar);
13   begin
14       Comportamento.Executar;
15   end;
16   procedure TPato.Comer(Comportamento: TComer);
17   begin
18       Comportamento.Executar;
19   end;
20   procedure TPato.Grasnar(Comportamento: TGrasnar);
21   begin
22       Comportamento.Executar;
23   end;
24  end.
```

Listagem 7. Simulando um pato que come

```
01  var
02   Pato: TPato;
03  begin
04   Pato := TPato.Create;
05   Pato.Comer(TComer.Create);
06  end.
```

Vamos começar pelos comportamentos dos patos de borracha. Eles não podem comer ou caminhar, mas podem grasnar. Como foi dito anteriormente, os patos de borracha emitem um ruído diferente do ruído que o pato real emite ao grasnar, então é necessário criar outra classe para simular o grasnado de um pato de borracha: eu preferi chamar esta nova classe de TGrasnarDoPatoDeBorracha (**Listagem 8**). Ela será muito parecida com a classe TGrasnar, pois tem o mesmo objetivo. A diferença entre elas é a implementação do método Executar().

Nota

Um risco no uso de injeção de dependência, principalmente quando definimos cadeias mais complexas de criação de objetos, é a referência circular, em geral de difícil detecção. As causas mais comuns são:

- Objetos gerados através de Constructor Injection que fazem referência mutuamente em seus parâmetros de um construtor, ou ainda, onde uma instância de uma classe é passada como um parâmetro para seu construtor.
- Objetos gerados por meio de Setter Injection que fazem referência uns aos outros.

Também será necessário criar mais duas novas classes para simular os comportamentos dos patos que não podem comer ou grasnar. Serão duas classes bem simples e parecidas com a classe TGrasnarDoPatoDeBorracha, veja na **Listagem 9** e **Listagem 10**.

Listagem 8. O "grasnar" de um pato de borracha

```
01  unit GrasnarDoPatoDeBorracha;
02  interface
03  uses Dialogs;
04  type
05   TGrasnarDoPatoDeBorracha = class
06   public
07    procedure Executar;
08   end;
09   implementation
10   procedure TGrasnarDoPatoDeBorracha.Executar;
11   begin
12       ShowMessage('Gueeeq... Gueeeq... Gueeeq...');
13   end;
14  end.
```

Listagem 9. Comportamento para simular um pato que não come

```
01  unit NaoComer;
02  interface
03  type
04   TNaoComer = class
05   public
06    procedure Executar;
07   end;
08   implementation
09   procedure TNaoComer.Executar;
10   begin
11       //Não faz nada por não poder comer
12   end;
13  end.
```

Listagem 10. Comportamento para simular um pato que não caminha

```
01  unit NaoCaminhar;
02  interface
03  type
04   TNaoCaminhar = class
05   public
06    procedure Executar;
07   end;
08   implementation
09   procedure TNaoCaminhar.Executar;
10   begin
11       //Não faz nada por não poder caminhar
12   end;
13  end.
```

Por sua vez, os patos eletrônicos podem caminhar e emitem o mesmo ruído que um pato real emite ao grasnar, mas não será necessário criar nenhuma nova classe para este pato. Ao invés disto serão aproveitadas as classes TGrasnar e TCaminhar que já existem para simular o pato real. Simples assim! No antigo design do Simulador de Patos não era possível reaproveitar comportamentos de outros patos, pois cada tipo de pato era uma classe fechada que não permitia ser reutilizada, a não ser que mais herança fosse criada.

Agora se tornam evidente outras vantagens de se usar a composição ao invés da herança, pois (graças à composição) é possível

reaproveitar os comportamentos dos patos reais para os patos eletrônicos, já que este dois tipos de patos emitem o mesmo ruído ao grasnar e caminham da mesma forma. Com isto conseguimos evitar a duplicação de código, adequando o projeto a mais um princípio: o princípio DRY – Don't Repeat Yourself.

As interfaces

A **Figura 7** mostra o design final do projeto, onde é possível notar que as novas classes criadas no tópico anterior não podem ser utilizadas porque a classe TPato está relacionada com as classes concretas TCaminhar, TGrasnar e TComer. Existem duas alternativas para corrigir este problema: usar herança ou interfaces.

Como você já aprendeu que a herança nem sempre tem o melhor custo benefício para o projeto, vamos esquecer essa ideia e partir para o uso das interfaces. Lembre-se de que a herança deve ser evitada sempre que for possível.

Nota do DevMan

O conceito Don't repeat yourself - em português "Não se repita" - defende que cada porção de conhecimento em um projeto tenha uma única representação. Colocá-lo em prática resulta em um projeto mais fácil de modificar, pois alterações em uma parte não levam a modificações em outras partes não relacionadas. Foram Andy Hunt e Dave Thomas que criaram e defenderam este conceito em seu livro The Pragmatic Programmer, publicado em 1999.

Pensando em evitar a herança, a segunda opção é usar interfaces. As interfaces funcionam quase da mesma forma que as classes abstratas. Com elas é possível definir um grupo de classes para cada comportamento de pato que será simulado, assim como seria feito com a herança. A diferença é que a herança obriga que as subclasses implementem somente os métodos abstratos da superclasse, enquanto que a interface obriga que todos os métodos sejam implementados.

As interfaces atuam como um contrato que determina o que as classes devem fazer. Quando uma classe implementa uma interface ela é obrigada a implementar todos os métodos declarados pela interface. As **Listagem 11**, **Listagem 12** e **Listagem 13** mostram as interfaces criadas para o Simulador de Patos.

Implementar uma interface é tão fácil quanto criar uma subclasse. Compare a **Listagem 14** com a **Listagem 8** e você verá que a única diferença está nas linhas 3 e 5, onde a interface IGrasnar é implementada na classe em TGrasnarDoPatoDeBorracha. Não foi necessário criar nenhum método novo, pois a classe já tinha o método Executar(), requerido pela interface.

Todas as classes de comportamentos foram alteradas para implementar as respectivas interfaces, mas eu escolhi não colocar o código de todas as classes no artigo já que esta é uma alteração bem simples e muito parecida com o que foi feito na **Listagem 14**. Você pode ver mais detalhes no exemplo do artigo disponível para download.

Para finalizar, os métodos Caminhar(), Grasnar() e Comer() da classe TPato devem ser atualizados para receber qualquer classe que implemente uma das novas interfaces, ao invés de ficar preso em uma implementação concreta como era antes. Veja na **Figura 8** como ficou o design do projeto depois de todas estas alterações.

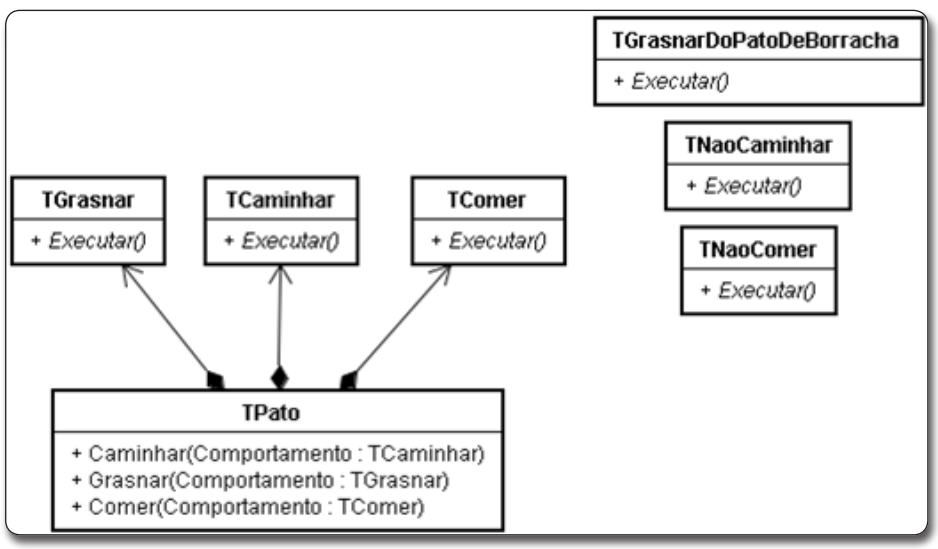


Figura 7. Design final: parte 1

Listagem 11. Interface IGrasnar

```
01  unit InterfaceGrasnar;
02  interface
03  Type
04      IGrasnar = interface
05    procedure Executar;
06   end;
07  Implementation
08  end.
```

Listagem 12. Interface IComer

```
01  unit InterfaceComer;
02  interface
03  type
04   IComer = interface
05     procedure Executar;
06   end;
07  implementation
08  end.
```

Listagem 13. Interface ICaminhar

```
01  unit InterfaceCaminhar;
02  interface
03  type
04   ICaminhar = interface
05     procedure Executar;
06   end;
07  implementation
08  end.
```

Listagem 14. Implementando a interface IGransar

```
01  unit GrasnarDoPatoDeBorracha;
02  interface
03  uses Dialogs, InterfaceGrasnar;
04  type
05   TGrasnarDoPatoDeBorracha = class
     (TInterfacedObject, IGrasnar)
06   public
07     procedure Executar;
08   end;
09   implementation
10   procedure TGrasnarDoPatoDeBorracha.Executar;
11   begin
12      ShowMessage('Gueeeq... Gueeeq... Gueeeq...');
13   end;
14  end.
```

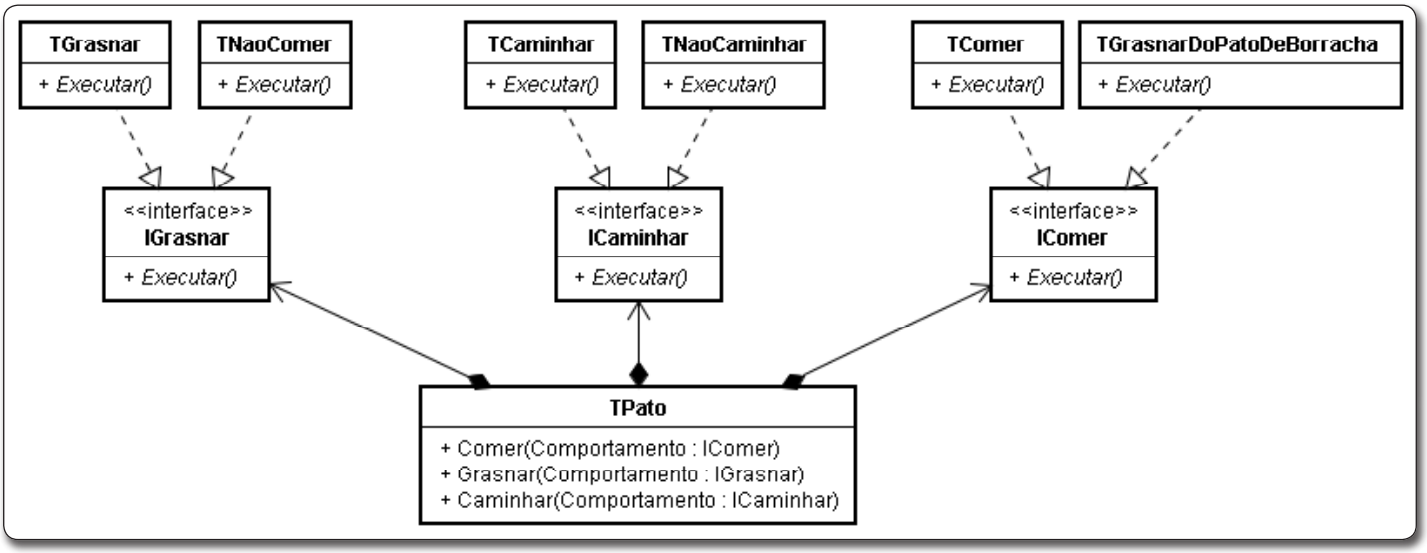


Figura 8. Design final: parte 2

Inversão de controle X Injeção de dependência

O Simulador de Patos passou por muitas modificações porque, na primeira versão ele era pouco escalável já que suas classes estavam fortemente ligadas. Aquele era um design bem simples de codificar, porém com poucas vantagens quando comparado com as consequências ruins que ele trazia.

Então, sem aumentar a complexidade do código, o design foi mudando pouco a pouco com objetivo de substituir a herança que foi a grande vilã da primeira versão, pela agregação. Seguindo poucos princípios da programação orientada a objetos, o Simulador de Patos tornou-se um projeto mais robusto e flexível, além de ficar mais intuitivo entender qual a função de cada classe.

O tempo todo o projeto esteve a caminho rumo à inversão de controle, pois a intenção era tirar da classe TPato o controle sobre seus comportamentos, que passariam a ser de outras classes específicas para os comportamentos de patos. Esta não é a única forma de usar a inversão de controle como já foi dito no início do artigo. A inversão de controle é nome que define qualquer técnica que inverte o fluxo normal em uma operação, como foi feito com a classe TPato onde é mais comum esperar que os comportamentos dos patos fiquem dentro da própria classe.

Mesmo separando os comportamentos em classes distintas, a classe TPato ainda precisa deles para poder simular um pato qualquer, por isto é criada uma dependência da classe TPato em relação às classes de comportamento. Acontece que esta é uma dependência flexível, pois é possível montar qualquer tipo de pato usando uma combinação diferente de comportamentos.

Para poder montar um pato, é necessário injetar o comportamento necessário e daí vem a Injeção de Dependência. Veja: a classe TPato depende das classes de comportamentos e para montar um pato os comportamentos desejados são injetados em tempo de execução em uma instância de TPato.

Quando se escuta falar em inversão de controle e injeção de dependência dá-se a impressão de que são técnicas super difíceis de serem implantadas, mas como você acabou de ver isto não é verdade. Bastou seguir alguns princípios de programação orientada a objetos para que injeção de dependência aparecesse naturalmente.

No Simulador de Patos foi feito injeção de dependência manual, o que deu um bom resultado, mas ainda existem mais vantagens em usar a injeção de dependências.

Container de dependências X Injeção automática

Criar dependências entre as classes para injetá-las posteriormente, quando for necessário, é a forma mais comum de usar a injeção de dependências, porém a injeção de dependência pode ser feita de outras formas.

Outra forma comum de usar a injeção de dependência é com o uso de containers de dependências. Nesta abordagem cria-se uma classe que servirá como container para armazenar as instâncias que serão injetadas posteriormente. A grande vantagem desta abordagem é a redução radical do uso de expressões como TMinhaClasse.Create, pois a criação de muitas das classes usadas pelo projeto fica a cargo do container. Toda classe registrada no container passa a ser criada por ele próprio, que pode inclusive controlar o tempo de vida da instância que será criada para cada classe registrada.

Um container de dependências é algo simples de codificar. Com alguns princípios de orientação a objetos e padrões de projetos é possível criar um bom container em pouco tempo, porém dificilmente você irá construir um, pois existem bons containers Open Source disponíveis na internet, um exemplo é o Delphi DI Container. Com ele você pode criar dependências usando Custom Attributes, veja um exemplo na **Listagem 15**, onde existe um atributo customizado na linha 4. O código mostrado nesta listagem fará com que o Delphi DI Container injete automaticamente a instância que será usada na propriedade ComportamentoDeComer.

Listagem 15. Injeção de dependências com Custom Attributes

```
01  type
02   TPato = class
03   public
04    [InjectByType(IComer)]
05    property ComportamentoDeComer: IComer;
06   end;
```

Além do Delphi Di Container, existe outros bons containeres, como por exemplo, o Emballo que tem uma grande vantagem para nós brasileiros: a documentação dele está em duas línguas, português e inglês. Até onde eu vi, ele não dá suporte ao uso dos atributos customizados para fazer injeção automática, mas a sua implementação é mais simples e a curva de aprendizado é menor quando comparado com o Delphi DI Container.

Além dos containeres ainda existem os Frameworks como o Delphi Spring Framework, que oferece mais recursos do que um simples container pode oferecer. O nome Spring vem de um famoso framework na comunidade Java conhecido por Spring Framework. O Delphi Spring Framework oferece muitas vantagens, mas sua documentação está em inglês.

Conclusão

A inversão de controle é um conhecimento obrigatório para qualquer programador que se preze e traz grandes vantagens sem esforços. Basta seguir as boas práticas como respeitar os princípios de programação orientada a objetos e implementar os padrões de projetos já comprovados.

Se você não tiver muita experiência ainda pode contar com a ajuda dos containers e frameworks para fazer a injeção automática por você. Não deixe de dar uma olhada nas ferramentas que citei e boa diversão.

Rafael Stavarengo
faelsta@gmail.com - http://stavarengo.com/blog
Programador de sistemas a 8 anos, integrante da equipe editorial da revista Clube Delphi. Domínio em Java, PHP, Object Pascal e UML. Sólido conhecimento em Design Patterns e metodologias ágeis. Graduado em Análise e Desenvolvimento de Sistemas pela UNIPAR.

Delphi DI Container
http://code.google.com/p/delphidicontainer/

Emballo
http://code.google.com/p/emballo/

Delphi Spring Framework
http://code.google.com/p/delphi-spring-framework/

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre ouutros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DevMedia

Usando o Padrão Singleton

Vitor Luiz Rubio

Singleton é um padrão de projeto bastante conhecido da GOF (Gang of Four) onde uma classe pode ter apenas uma instância. Ele tenta garantir que a classe seja instanciada uma única vez e fornece um ponto de acesso global a ela.

Alguns componentes geralmente modelados como singletons são: camadas de acesso a dados, serviços de comunicação, como envio de e-mails, o objeto de configuração do sistema como um todo e toda e qualquer informação que tenha uma fonte única e que deva ser compartilhada por todas as outras classes.

Especialmente se uma classe de serviço ou de sistema é muito grande e custosa para se instanciar o uso de singleton faz com que durante toda a execução do sistema a classe seja instanciada uma única vez.

Classes que usam recursos escassos e / ou que não podem ser compartilhados, por exemplo, no caso de leitura e escrita em um arquivo de texto, apenas uma instância da classe teria acesso para leitura e escrita por vez. Outras tentativas de abrir um mesmo arquivo para leitura e escrita resultariam em um deadlock.

O uso de singletons deve ser justificado, visto que recentemente muitos têm considerado o singleton um anti-pattern. Afinal muitos dos problemas que alguns programadores costumam resolver usando singletons são possivelmente solucionáveis com injeção de dependência.

Singletons não devem ser usados como variáveis globais, mas sim como objetos que mantêm um estado comum e público a todas as classes do sistema. É o tipo de padrão de projeto que, na dúvida, é melhor não usar. Ele não deve ser usado como uma "variável global orientada a objeto", nunca.

É realmente muito raro singletons serem necessários, mas tenho visto muitos membros da comunidade implementar singletons onde ele não é estritamente necessário, e o pior, implementar de maneira errada. Em alguns casos a implementação errada não causa nenhum impacto no sistema, mas em muitos casos elas podem causar erros de Access violation e memory leaks.

Nota

Se o singleton possui apenas métodos, mas não atributos, muitas pessoas implementam o singleton como uma classe com vários métodos de classe. Isso não é recomendável, pois viola vários princípios da orientação a objeto (regras de negócio espalhadas que não fazem parte de nenhuma classe). E mesmo que a classe não precise ser instanciada, ela ainda pode ser, embora neste caso não faça muita diferença. Você pode ter várias classes dependendo do seu singleton, mas é imprescindível que o singleton não dependa de nenhuma outra classe, pois isso causaria uma bagunça muito difícil de se consertar.

Resumo DevMan

De que trata o artigo:

O artigo tratará do padrão de projeto Singleton, mostrando como implementá-lo e que erros comuns devem ser evitados. Também serão abordadas certas peculiaridades provindas dos novos recursos da Delphi Language que podem ajudar na construção de Singletons.

Em que situação o tema é útil:

Singleton é uma classe que deve ter apenas uma instância. É muito mais usado para representar classes de sistema e de serviços gerais do que classe de negócio/domínio. Em uma arquitetura multicamadas e/ou em DDD eles são usados para representar conexões ao banco de dados, servers, serviços como envio de e-mail, SMS entre outras funções. Esse tema é útil principalmente para as empresas que vendem frameworks para que outras possam desenvolver sistemas mais rapidamente. Se uma classe deve ser um singleton os programadores do framework devem assegurar isso.

Singleton:

Um padrão de projeto muito comum e até mesmo visto por alguns como um "anti-padrão" é o singleton. Singletons são usados para representar serviços de sistema, e não serviços de negócio. São apropriados para instâncias ou seções de banco de dados, mecanismos de persistência, envios de e-mail, fila de impressão e assim por diante. Muitas vezes os singletons são implementados de maneira errada, até mesmo perigosa. Não existe maneira "nativa" de se criar um singleton em Delphi, mas existem alternativas elegantes.

Para explicar o porquê disso, deve-se raciocinar: se os singletons não devem ser criados, eles também não devem ser destruídos, pois uma vez destruídos quem os recriará? E pior: as referências a ele (variáveis que o contém/apontam para ele) passam a ser inválidas no momento em que ele é destruído. Inválidas, porém diferentes de nil. Criar um singleton que já tem uma instância também é problemático. Depois que a segunda instância é criada e atribuída ao objeto global a referência ao objeto anterior é perdida. Sem referência, como o objeto será destruído? Causa-se assim um memory leak.

Um singleton também não deveria ser atribuível a outras variáveis ou passado como parâmetro para um método, pois não dá para garantir muito o que se fará de errado com ele.

Na seção de links desse artigo encontra-se uma série de URLs de blogs e fóruns discutindo sobre o uso de singletons e porque eles são considerados um anti-pattern.

Note que o que está sendo questionado é o uso indiscriminado do singleton como variável global. Para programadores que estão saindo de ambientes procedurais ou orientados a eventos e tendo suas primeiras incursões na programação OO pura os singletons podem parecer uma atraente alternativa às variáveis públicas globais.

Os leitores deste artigo poderão se perguntar: "mas para que fazer um artigo sobre esse assunto se não há consenso com relação aos seus benefícios?". A resposta é simples, independente do uso correto do singleton saber como criá-lo é um conhecimento interessante para habitar a "caixa de ferramentas" de qualquer programador, mas o mais interessante é que o modo de se criar tal padrão pode mostrar detalhes interessantes do próprio Delphi.

Maneiras de implementar

A maneira mais comum de se implementar um singleton é por ter uma variável pública na seção implementation da unit onde a classe reside a fim de guardar uma instância da classe a ser criada, e colocar na seção interface uma função estática pública global que retorne esta instância caso criada. Se a variável for nil então a função primeiro a cria e depois a retorna.

Uma melhoria nesse tipo de implementação seria transformar a variável pública global em uma class var (campo estático de classe) que seria limitada unicamente ao escopo da classe, e uma class function (método de classe) para substituir a função estática pública global.

As desvantagens claras dessas duas abordagens é que elas não estão protegidas caso um programador desavisado que não tenha lido o "manual" resolva usar o método create, ou pior, o Destroy.

A **Listagem 1** mostra uma comparação entre essas duas formas. A seção de implementação foi omitida porque além desse exemplo ser teórico, é bastante trivial. No código-fonte que acompanha esta edição podem ser analisadas implementações completas de singletons.

Todos os testes desse artigo são executados com a opção ReportMemoryLeaksOnShutdown := true; no arquivo .dpr. Isso mostrará uma mensagem caso ocorram memory leaks, ou seja, caso o singleton tenha algum objeto criado que não foi destruído.

Caso faça o teste criando instâncias desses objetos perceberá que, nas duas maneiras, você pode sim criar várias instâncias do objeto em questão caso teimosamente use o construtor "Create", mas apenas uma será destruída na seção finalization: a que tiver a referência válida. As outras instâncias que estiverem na memória causarão memory leaks.

O objeto será destruído ao se finalizar a aplicação, caso exista, mas não é criado sob a seção initialization, é criado sob demanda caso se use a função pública estática global que o retorne. Ela terá a missão de verificar se o objeto já existe ou não e criá-lo sob demanda. Ou seja, se por descuido o objeto for criado mais de uma vez, ele só será destruído uma única vez.

O que muitos programadores tentam fazer para proteger essas classes de "criação indiscriminada" é colocar private no método create. O problema é que o método create não pode ser "escondido" simplesmente por colocá-lo como private. Na verdade nenhum método público pode.

Listagem 1. Singletons básicos

```
type
 TSingleton1 = class
 private
  FCreationTimeStamp: string;
 public
  function DoSomethingImportant: string;
  constructor Create;
 end;
 function Singleton1Instance: TSingleton1;
type
 TSingleton2 = class
 private
  class var _Singleton2Instance: TSingleton2;
  FCreationTimeStamp: string;
 public
  class  function Singleton2Instance: TSingleton2;
  function DoSomethingImportant: string;
  constructor Create;
 end;
```

Cabe aqui uma breve explicação sobre o método create. Ele atua como uma class function para criar uma instância de um objeto, mas não é ele realmente que o cria.

Métodos públicos sempre serão visíveis pelos seus ancestrais, por mais que se tente sobrescrevê-lo com um private. Se uma classe A tem um método "foo" público, a classe B é filha de A e sobrescreve o método "foo" com um método privado e a classe C é filha de B e não tem esse método "foo", mesmo assim ele será visível: será visível a implementação de A. Isso quer dizer que todo objeto tem um constructor sem parâmetros ou que não faz nada, que é o constructor herdado de TObject.

Por outro lado métodos protected podem ser acessados por subclasses ou dentro de uma mesma unit, mas não por classes fora da família. Esses métodos podem ser publicados definitivamente ou escondidos definitivamente. Se o constructor create fosse um método protected em TObject seria possível escondê-lo.

Alguns contornam essa situação por disparar um exception logo na criação do objeto, caso já exista uma instância. Isso sem dúvida serve para alertar definitivamente os usuários do seu framework que estão usando essa classe de modo errado, mas é uma solução um tanto quanto radical e ainda pode causar memory leaks ou outros erros caso a exception não seja a primeira coisa que o constructor faça. Por exemplo, se a exception for disparada depois da criação de um objeto interno então criou-se um leak.

Por fim há quem sobrescreva o método NewInstance, quem use construtores strict private e quem faça um misto de tudo isso.

Há situações onde uma classe de uma unit comumente usada em vários projetos deveria ser um singleton apenas em um sistema ou em um determinado contexto, mas não deveria em outros. Para esses casos deve existir uma maneira de se controlar o número de instâncias de uma classe qualquer e assegurar que só tenha uma.

Nota

Nada impede que se crie constructors com outro nome que sejam private, mas mesmo assim o constructor create ainda será visível por classes que a usem e "aparece" no code completion do IDE do Delphi.

Singletons no Delphi e herança

Como dito, objetos singletons são aqueles que têm apenas uma instância no sistema. No Delphi, e em programas feitos por ele, os objetos Screen e Application podem ser considerados singletons, afinal possuem apenas uma instância.

A melhor maneira de se proteger ou isolar um singleton é com a documentação. Você não vê por aí nenhum exemplo de pessoa tentando criar um objeto do tipo TApplication ou TScreen, mas apenas usando os objetos publicamente criados.

A **Listagem 2** apresenta um exemplo do que pode acontecer caso se violem essas regras.

Listagem 2. Application é um singleton da classe TApplication

```
var app: TApplication;
begin
  app := TApplication.Create(nil);
  app.Initialize;
  app.Run;
  app.Terminate;
end;
```

O Delphi usa variáveis globais para os singletons da VCL, mas não oferece nenhum tipo de proteção. Ele assume que o programador usará esses objetos corretamente.

Embora o código da **Listagem 2** esteja errado, nada impede que ele seja executado. Se for executado ocorrerá um runtime error e a aplicação fechará. Só pode haver uma instância de TApplication e uma de TScreen, que são as variáveis públicas globais Application e Screen, respectivamente.

Herança de singletons pode ser um problema, considere o seguinte fato sobre POO: uma instância de uma classe filha também pode ser considerada uma instância válida de sua classe pai, mas o contrário não é verdadeiro. Logo, se você já tem um objeto instanciado de uma classe que deveria ser um singleton você não deveria nunca instanciar um objeto de uma subclasse, ou classe "neta" desta.

Outro motivo pelo qual não faz muito sentido usar herança em singletons é que esses padrões geralmente são usados para representar recursos ou representar a camada mais baixa, mais interna do sistema. Sendo assim não faz sentido ter algo "derivado" dessa camada mais interna.

Mesmo assim alguns modelos de singleton podem ser herdados, mas não podem ser usados em conjunto com a classe base. Outros modelos podem implementar internamente, na classe mais básica, um hashlist para poder ter uma, e apenas uma, instância de cada classe na hierarquia. Não vejo muita utilidade nisso, mas existem meios de se implementar.

Algumas características da POO no Delphi

Se um método é marcado como protected ele não pode ser executado "de fora" da classe, assim como o private, mas ele pode ser executado por descendentes da classe.

Outra coisa que um descendente pode fazer é sobrescrever o método protected. Se a classe derivada redeclara o método como public ele passa a ser público e visível por todos. Se ela o redeclara como private o método será escondido, não podendo ser visto de fora e nem pelos descendentes.

É por isso que não se pode ocultar totalmente o Create, ou sobrescrevê-lo como private: porque ele foi declarado como public já em TObject.

A **Listagem 3** ilustra essas características. As classes a seguir têm métodos públicos que não podem ser escondidos em seus descendentes e métodos protegidos que podem. Outro ponto importante é que um método marcado como virtual poderá ser sobrescrito com override em uma classe derivada, e será o método executado, mesmo que o tipo da variável que contenha o objeto seja do tipo da classe base. Se ele não fosse marcado como virtual, então o método da classe derivada iria sobrescrever o da classe base, mas numa variável do tipo base o método é que seria executado. Tente tirar a palavra virtual e override dos métodos e verá que em variáveis do tipo TClasseA será executado um MetodoQueNaoPodeSerEscondido idêntico ao da classe A, sem os acréscimos das classes B e C, mesmo que a instância seja do tipo B e C. Lembre-se que uma variável de um tipo "base" pode "carregar" objetos de qualquer um dos seus descendentes, e que uma variável do tipo Tobject pode apontar para um objeto de qualquer tipo. Ao final da **Listagem 3** encontra-se um exemplo de uso dessas classes.

Tentar ocultar o constructor Create por colocá-lo como private ou strict private será um exercício esclarecedor para os próximos exemplos práticos, e justificará uma série de medidas tomadas nas implementações dos singletons 4 e 5

Protegendo o Singleton

Aproveitando o código das implementações da **Listagem 1**, pode-se tentar proteger o singleton por colocar uma exception na criação, mas isso como já dito, além de radical não protege completamente contra o leak, pois uma nova instância é criada bem antes da exception ser disparada.

Quando o controle da existência prévia do singleton for feito pelo método estático ou pelo método de classe não há muito problema em se criar uma segunda instância do singleton, contanto que ela seja armazenada em uma variável local ou num bloco try ... finally e destruída logo em seguida, no mesmo contexto onde foi criada. O singleton poderia ser criado e destruído num mesmo contexto sem causar leak.

O problema seria pior caso o controle (checagem e atribuição da variável de instância) fosse feito no constructor. Na maioria das vezes, no entanto, os singletons são usados para alocar recursos que não deveriam ser compartilhados em hipótese alguma ou até servir como um cache, ou acesso rápido a recursos somente leitura de carregamento muito custoso e baixa taxa de alteração. Ou seja, contextos em que não faz sentido uma segunda instância.

Disparar uma exceção durante uma tentativa de segunda criação do singleton pode proteger apenas contra isso: uma segunda instância, mas não protege contra sua destruição. Singletons não deveriam ser destruídos já que não podem ser criados, principalmente se o overhead para se criar um é alto. Eles deveriam ser destruídos apenas na finalização da aplicação.

Listagem 3. Exemplos de métodos protected e public

```
unit metodos;
interface
uses
 dialogs;
type
 TClasseA = class
 protected
  procedure MetodoQueSeraPublicado; virtual;
  //pode ser publicado porque é
  protected
  procedure MetodoQueSeraEscondido; virtual;
  //pode ser escondido porque é
  protected
 public
  procedure MetodoQueNaoPodeSerEscondido; virtual;
  //não pode ser escondido
  porque é public
 end;
 TClasseB = class(TClasseA)
 private //também pode ser strict private
  procedure MetodoQueSeraEscondido; override;
  procedure MetodoQueNaoPodeSerEscondido; override;
 public
  procedure MetodoQueSeraPublicado; override;
 end;
 TClasseC = class(TClasseB);
implementation
 { TClasseA }
 procedure TClasseA.MetodoQueNaoPodeSerEscondido;
 begin
    ShowMessage(ClassName + '.MetodoQueNaoPodeSerEscondido');
 end;
 procedure TClasseA.MetodoQueSeraEscondido;
 begin
    ShowMessage(ClassName + '.MetodoQueSeraEscondido');
 end;
 procedure TClasseA.MetodoQueSeraPublicado;
 begin
    ShowMessage(ClassName + '.MetodoQueSeraPublicado');
 end;
 { TClasseB }
 procedure TClasseB.MetodoQueNaoPodeSerEscondido;
 begin
    inherited;
    ShowMessage(ClassName + ' --> mais código pode ser posto aqui');
 end;
 procedure TClasseB.MetodoQueSeraEscondido;
 begin
    inherited;
    ShowMessage(ClassName + ' --> mais código pode ser posto aqui');
 end;
procedure TClasseB.MetodoQueSeraPublicado;
begin
 inherited;
 ShowMessage(ClassName + ' --> mais código pode ser posto aqui');
end;
end.
{....}
procedure TForm1.Button7Click(Sender: TObject);
var
 ca: TClasseA;
 cb: TClasseB;
 cc: TClasseC;
 cgenerico: TClasseA;
begin
 ca := TclasseA.Create;
 cb := TclasseB.Create;
 cc := TclasseC.Create;
 ca.MetodoQueNaoPodeSerEscondido;
 //ca.MetodoQueSeraPublicado; //não pode ser chamado daqui pois é protected
 //ca.MetodoQueSeraEscondido;
 //não pode ser chamado daqui pois é protected
 cb.MetodoQueSeraPublicado; //pode ser chamado pois nesta classe é public
 //cb.MetodoQueSeraEscondido; //não pode ser chamado porque é private
 cb.MetodoQueNaoPodeSerEscondido;
 //pode ser chamado, mas é o de TClasseA, não
 há code completion para ele
 //mesmo caso de CB, mas MetodoQueNaoPodeSerEscondido agora tem code completion
 cc.MetodoQueSeraPublicado;
 cc.MetodoQueNaoPodeSerEscondido; //tem code completion, mas continua
  sendo o de TClasseA
 ca.Free;
 cb.Free;
 cc.Free;
 ShowMessage('Testando MetodoQueNaoPodeSerEscondido de cgenerico');
 cgenerico := TclasseA.Create;
 cgenerico.MetodoQueNaoPodeSerEscondido;
 cgenerico.Free;
 cgenerico := TclasseB.Create;
 cgenerico.MetodoQueNaoPodeSerEscondido;
 cgenerico.Free;
 cgenerico := TclasseC.Create;
 cgenerico.MetodoQueNaoPodeSerEscondido;
 cgenerico.Free;
end;
```

No Delphi XE disparar uma exceção no começo de um contructor não permitirá que a instância seja criada e retornará nil. A **Listagem 4** mostra como ficaria a classe TSingleton3 com essa proteção no constructor.

Temos apenas a interface, pois mais modificações podem ser feitas na implementação. Colocar uma exception no constructor não o isola e nem impede que seja usado. Se alguém tentar criar várias instâncias do objeto sem usar o método de classe GetInstance pode fazer usando diretamente o próprio constructor. Como a variável _Instance vai ser sempre nil múltiplas instâncias poderiam ser criadas. Para se evitar isso pode – se atribuir a _Instance a "variável" self, uma referência ao próprio objeto sendo criado.

Como será visto mais adiante, a essa altura o objeto já existe, visto que o construtor nada mais é do que um inicializador. Considere o seguinte código:

```
Class function TSingleton3.GetInstance: TSingleton3;
begin
 if TSingleton3._Instance = nil then
  TSingleton3._Instance := TSingleton3.Create;
 Result := TSingleton3._Instance;
end;
```

Listagem 4. Interface do TSingleton 3

```
TSingleton3 = class
private
  class var _Instance: TSingleton3;
  FCreationTimeStamp: string;
public
  class  function GetInstance: TSingleton3;
  function DoSomethingImportant: string;
  constructor Create;
end;
```

Ele poderia ser reduzido a:

```
class function TSingleton3.GetInstance: TSingleton3;
begin
 if TSingleton3._Instance = nil then
  TSingleton3.Create;
 Result := TSingleton3._Instance;
end;
```

Isso porque o create já atribui a instância do singleton ao campo _Instance. Mas por motivos de clareza de código isso não será feito. Falta ainda impedir que o destructor cause algum dano. Há duas maneiras de se fazer isso: 1) o destructor atribui nil à

variável _Instance para que numa tentativa de se criar o objeto ele possa ser criado com segurança, já que _Instance será nil. 2) Pode-se simplesmente impedir a destruição do objeto. Isso tornará difícil desalocar a memória usada por ele.

Deixar que o objeto seja criado indiscriminadamente pode ser um erro se o custo para criá-lo for elevado, como uma consulta pesada em um banco de dados na web, ou o consumo de um recurso em um computador remoto.

O método Free chama o destructor original de TObject e finaliza o objeto. Mas não é o Destructor que desaloca a memória. Isso é trabalho do FreeInstance. O método free original de TObject é assim:

```
procedure TObject.Free;
begin
  if Self <> nil then
    Destroy;
end;
```

E ele pode ser sobrecarregado para que não destrua o objeto em certas ocasiões, por exemplo, quando o _Instance for diferente de nil. Assim para executar free seria necessário atribuir nil a _Instance primeiro, como é mostrado a seguir. Mas isso não impediria de se executar o método Destroy:

```
procedure TSingleton3.Free;
begin
  if _Instance = nil then
    inherited Free;
end;
```

Isolar uma classe do mundo externo é bom não só para singletons, mas também para classes que não devem ter seus construtores expostos, ou que possuem construtores especiais com vários parâmetros ou para classes que devem ser criadas apenas com factory methods de outras classes.

Simplesmente não é possível que uma classe não possua um construtor ou o deixe "invisível". O construtor sem parâmetro, chamado na literatura de "parameterless", pode ser sobrecarregado com um construtor com parâmetros. Mas se forem necessários dois construtores com parâmetros diferentes marcados com "overload", o próprio construtor sem parâmetros será então uma opção válida de compilável.

Alguns programadores, como saída, ocultam apenas o construtor sem parâmetros por criar uma classe base onde o construtor sem parâmetro é strict private. Depois criam outras classes como descendentes desta classe base e criam construtores com parâmetros. Assim a classe pode ter vários construtores sem parâmetros e nenhum "default".

Ou seja, um construtor default (parameterless) pode ser escondido com a criação de um construtor com parâmetros, mas nenhuma classe pode ter construtores invisíveis ou privados parameterless, pois o constructor de TObject é público e visível.

Um bom exemplo disso é a classe TForm. Ela tem um constructor que precisa de um parâmetro (aOwner) e não aceita ou não possui constructor Create() sem parâmetros. Isso se dá porque em TPersistent já foi criado o construtor Create(aOwner: TObject) que sobrescreve e oculta o Create sem parâmetros.

Não importa o quanto você tente ocultar um constructor, seja com parâmtros ou não, algum deles sempre será visível. Se houver mais de um constructor Create com parâmetros (overload), sempre estará visível e disponível o constructor original, primordial, da classe TObject, e esse não pode ser ocultado. A única opção seria criar um único construtor com parâmetro que o sobrescrevesse.

O perigo maior reside no fato de que o constructor primordial não inicializa as variáveis ou outros objetos necessários à classe. Se forem strings, estarão vazias, se forem outros objetos (composição) não serão criados, permanecendo nil e causando access violations.

Apenas a "casca vazia" da classe será criada, sem nenhum dos outros objetos do qual ela precisa e com todos os fields primitivos com seus valores default. Isso se dá porque o constructor não foi declarado como protected na classe base, TObject. Um método protected pode ser publicado se a classe filha o colocar em public, ou escondido se a classe filha o colocar em private, mas um método público não pode ser alterado. Qualquer tentativa de esconder um método público numa classe derivada torna visível o método "ancestral". Além disso, o Create é um método de classe estático que tem por responsabilidade chamar o NewInstance.

Uma classe sempre tem um constructor visível, mesmo que ele não faça nada. Assegure-se que seu construtor faça o necessário e que a versão correta seja chamada.

Se o constructor for colocado como private ou strict private ele não aparecerá no "code completion" do Delphi quando você pressionar ctrl+espaço, e nem terá qualquer informação ou ajuda da IDE sobre o construtor. Mesmo assim ele pode ser chamado, basta digitá-lo e o programa compilará.

Já que um construtor com parâmetros pode isolar e esconder o construtor "parameterless" então é isso que será feito no singleton 4 para assegurar que o construtor "casca vazia" do TObject não seja nunca executado. Assim, seguramente será usado o construtor "tratado" que precisa de parâmetros, dispara uma exceção na tentativa de criar uma segunda instância e atribui Self a _Instance.

Isso fará com que a linha *Memo1.Lines.Add(TSingleton4.Create.DoSomethingImportant);* não execute e resulte em erro de compilação. Mas ainda não está bom. No singleton 3 estamos usando a seção finalization para destruir o objeto. Podemos fazer isso da maneira mais OO possível usando um Class destructor.

Class constructors e class destructors são construtores e destruidores estáticos de classe, e correspondem aos static constructors ou shared constructors do .Net (C# e VB.Net).

Esses construtores e destruidores de classe servem para inicializar (ou destruir/desalocar) variáveis, campos e atributos estáticos, de classe.

Como a instância de singleton _Instance é uma "class var", que é criada sob demanda pelo método GetInstance (lazy loading ou lazy binding), ela deve ser destruída e liberada da memória em algum ponto na destruição do sistema. Isso pode ser feito via class destructors.

Class constructors e class destructors podem ser private, pois eles nunca são chamados ou invocados por um programador ou por outro objeto. Eles são executados automaticamente na inicia-

lização ou finalização do sistema. O class constructor é invocado antes da classe ser usada pela primeira vez e o class destructor é chamado depois que a classe não está mais sendo usada. Eles são o equivalente OO às seções initialization e finalization, só que muito mais elegantes e poderosos.

A **Listagem 5** mostra o singleton 4 inteiro, com destructor de classe e constructor com um parâmetro, que esconde o constructor primordial.

O singleton 4 ficou bem melhor que os anteriores, mas ainda pode melhorar. E se fosse possível executar o create quantas vezes quiser, indiscriminadamente, mas ele nunca criar uma segunda instância? Há meios de fazer o create retornar sempre a mesma instância, mas para isso deve-se esclarecer como os construtores funcionam "por dentro".

O constructor

Construtores começam com a palavra chave Constructor no lugar de function ou procedure. Como é um método chamado a partir do nome da classe então pode-se dizer que é um método de classe, afinal seu uso é TNomeClasse.Create(parametros). E como o método Create retorna uma instância de um objeto, internamente ele é uma function, cujo retorno é do tipo da classe onde está contido.

Apesar de tudo isso os métodos marcados com a palavra chave "constructor", mesmo retornando uma instância do objeto e fazendo as devidas inicializações, são apenas "atalhos" de sintaxe, facilidades implementadas no compilador para tornar a linguagem mais legível. Quando se cria um objeto qualquer quem faz todo o "trabalho sujo" como alocação de memória e a real criação do objeto é o método NewInstance. O constructor apenas aproveita o resultado que NewInstance dá.

Da mesma forma o Destructor não desaloca o objeto, apenas executa as rotinas de "limpeza" destruindo outros objetos dos quais ele depende ou que ele possui internamente. Por exemplo, se sua classe tem internamente um TStringList, esse é instanciado no construtor, e destruido no destrutor.

Analogamente à dupla Create/NewInstance, quem faz o "trabalho sujo" para o Destroy é o FreeInstance. Este tem a finalidade de desalocar memória.

Se o controle da criação e da destruição de um objeto fosse feita no NewInstance e no FreeInstance então seria possível impedir que um objeto fosse criado ou destruído. Seria possível até chamar o Create várias vezes e mesmo assim apenas uma instância ser criada.

Esse singleton 5 da **Listagem 6** faz isso. O constructor Create pode ser chamado indiscriminadamente, várias vezes, e mesmo assim a instância retornada é a mesma.

Pode-se ver aqui que o método sobrecarregado NewInstance usa o mesmo esquema de lazy loading que os métodos para obter a instância dos exemplos anteriores. Se _Instance é nil então a instância corrente é criada simplesmente executando-se o inherited NewInstance, ou seja, o NewInstance ancestral. Se _Instance é diferente de nil então o NewInstance original não será criado, mas será retornada a instância que já existe em _Instance. Como o retorno de NewInstance é repassado, pelo compilador, para o constructor create, então qualquer invocação à create ou à NewInstance sempre retornará o mesmo objeto criado em _Instance, e isso pode ser comprovado tanto pela data de criação do objeto como pelo seu "endereço" na memória. A **Listagem 7** mostra como essa implementação pode ser usada e abusada com segurança.

É possível criar, destruir, fazer o que quiser com esse singleton, e sempre haverá apenas uma instância, a mesma criada da

Listagem 5. Singleton 4 com destructor de classe

```
unit singleton4;
interface
uses sysutils;
type
 Tsingleton4 = class
 private
   class var _Instance: Tsingleton4;
   FCreationTimeStamp: string;
 class destructor Destroy;
 public
   class  function GetInstance: Tsingleton4;
   function DoSomethingImportant: string;
   procedure Free;
   constructor Create(Teste: string); overload;
 end;
implementation
 { TSingleton1 }
 class  function Tsingleton4.GetInstance: Tsingleton4;
 begin
 if Tsingleton4._Instance = nil then
     Tsingleton4._Instance := Tsingleton4.Create('Construtor 1');
 Result := Tsingleton4._Instance;
 end;
constructor Tsingleton4.Create(Teste: string);
begin
if Tsingleton4._Instance <> nil then
    raise Exception.Create('Não pode haver mais de uma instância dessa classe!');
  _Instance := self;
FCreationTimeStamp:= DateTimeToStr(now) + ' com a mensagem ' + Teste;
end;
class destructor Tsingleton4.Destroy;
begin
if Tsingleton4._Instance <> nil then
  Tsingleton4._Instance.Destroy;
end;
function Tsingleton4.DoSomethingImportant: string;
begin
  Result := 'Instância de ' + Self.ClassName + ' Criada em ' +
  FCreationTimeStamp + ' no endereço ' + IntToStr(integer(self));
end;
procedure Tsingleton4.Free;
begin
if _Instance = nil then
  inherited Free;
end;
end.
```

Listagem 6. Singleton 5, sobrescrevendo NewInstance

```
unit singleton5;

interface

uses sysutils;

type
 Tsingleton5 = class
 strict private
   class var _Instance: Tsingleton5;
   class var _PodeDestruir: boolean;
   FCreationTimeStamp: string;
   class destructor Destroy;
   class constructor Create;
 public
   class function GetInstance: Tsingleton5;
   class function NewInstance: TObject; override;
   procedure FreeInstance; override;
   constructor Create;
   function DoSomethingImportant: string;
 end;

implementation


 { TSingleton1 }

 class  function Tsingleton5.GetInstance: Tsingleton5;
 begin
   if Tsingleton5._Instance = nil then
     Tsingleton5._Instance := TSingleton5.Create;
  Result := Tsingleton5._Instance;
 end;

 class constructor Tsingleton5.Create;
 begin
   Tsingleton5._PodeDestruir := false;
 end;

class destructor Tsingleton5.Destroy;
begin
  Tsingleton5._PodeDestruir := true;
  if Tsingleton5._Instance <> nil then
    Tsingleton5._Instance.Destroy;
end;

class function Tsingleton5.NewInstance: TObject;
begin
  if _Instance = nil then
    _Instance := (inherited NewInstance as Tsingleton5);
  Result := _Instance;
end;

constructor Tsingleton5.Create;
begin
  FCreationTimeStamp:= DateTimeToStr(now);
end;

function Tsingleton5.DoSomethingImportant: string;
begin
  Result := 'Instância de ' + Self.ClassName + ' Criada em ' +
  FCreationTimeStamp + ' no endereço ' + IntToStr(integer(self));
end;

procedure Tsingleton5.FreeInstance;
begin
  //destruirá apenas quando for permitido
  if _PodeDestruir then
  begin
    inherited FreeInstance;
    _Instance := nil;
  end;
end;
end.
```

primeira vez que foi necessária. E não haverá memory leak porque a instância padrão será destruida no destructor de classe.

Singleton com generics

Uma maneira interessante de transformar qualquer classe num singleton e controlar quantas instâncias são criadas é encapsular essa classe com uma outra genérica que cuidará das insâncias. Para isso o parâmetro genérico deve ser a classe a ser transformada em singleton.

Para que o parâmetro genérico seja criado dentro de um método da classe "wrapper", é necessário que a constraint "constructor" seja passada à classe wrapper, assim ela entenderá que o parâmetro genérico é uma classe que contém um constructor.

A classe TSingleton6<T> da **Listagem 8** pode receber qualquer tipo de classe que tenha um construtor parameterless padrão e criá-lo sob demanda com o método GetInstance, controlando para que haja apenas uma instância.

Como exemplo foram criadas nesta mesma unit as classes Tfoo e Tbar que não são singletons, mas são classes totalmente "normais" como outra qualquer.

Essa abordagem não impede que as classes Tfoo e Tbar sejam criadas por meios convencionais, mas é uma maneira elegante de transformar qualquer possível classe em um singleton e controlar suas instâncias. É mais "bonito" do que o uso de variáveis públicas globais como o Delphi faz com Application. A **Listagem 9** mostra como usar o singleton 6.

Listagem 7. Usando o singleton 5

```
procedure TForm1.Button5Click(Sender: TObject);
begin
   //pode usar à vontade
   Memo1.Lines.Add( TSingleton5.GetInstance.DoSomethingImportant);
   //criar à vontade
   Memo1.Lines.Add(TSingleton5.Create.DoSomethingImportant);
   Memo1.Lines.Add(TSingleton5.Create.DoSomethingImportant);
   Memo1.Lines.Add(TSingleton5.Create.DoSomethingImportant);
   Memo1.Lines.Add(TSingleton5.Create.DoSomethingImportant);
   Memo1.Lines.Add(TSingleton5.Create.DoSomethingImportant);
   Memo1.Lines.Add(TSingleton5.Create.DoSomethingImportant);
   //destruir à vontade
   TSingleton5.GetInstance.Free;
   TSingleton5.GetInstance.Free;
   TSingleton5.GetInstance.Free;
   TSingleton5.GetInstance.Free;
   TSingleton5.GetInstance.Free;
   //e continua funcionando
   Memo1.Lines.Add( TSingleton5.GetInstance.DoSomethingImportant);
end;
```

Listagem 8. Singleton 6 com generics

```
unit singleton6;
interface
uses
 sysutils;
type
 TSingleton6<T: class, constructor> = class
 strict private
  class var _Instancia: T;
  class destructor Destroy;
 public
  class function GetInstance: T;
 end;
 TFoo = class
 strict private
  FCreationTimeStamp: string;
 public
  constructor Create;
  function DoSomethingImportant: string;
 end;
 TBar = class
 strict private
  FCreationTimeStamp: string;
 public
  constructor Create;
  function DoSomethingImportant: string;
 end;
implementation
    { TSingleton6<T> }
    class destructor TSingleton6<T>.Destroy;
    begin
    if TSingleton6<T>._Instancia <> nil then
        TSingleton6<T>._Instancia.Free;
    end;
    class function TSingleton6<T>.GetInstance: T;
    begin
    if TSingleton6<T>._Instancia = nil then
        TSingleton6<T>._Instancia := T.Create;
    Result := TSingleton6<T>._Instancia;
    end;
    { TFoo }
    constructor TFoo.Create;
    begin
        FCreationTimeStamp:= DateTimeToStr(now);
    end;
    function TFoo.DoSomethingImportant: string;
    begin
        Result := 'Instância de ' + Self.ClassName + ' Criada em ' +
        FCreationTimeStamp + ' no endereço ' + IntToStr(integer(self));
    end;
    { TBar }
    constructor TBar.Create;
    begin
        FCreationTimeStamp:= DateTimeToStr(now);
    end;
    function TBar.DoSomethingImportant: string;
    begin
        Result := 'Instância de ' + Self.ClassName + ' Criada em ' +
    FCreationTimeStamp + ' no endereço ' + IntToStr(integer(self));
    end;
end.
```

Listagem 9. Usando o singleton 6

```
procedure TForm1.Button6Click(Sender: TObject);
begin
 //uso correto
 Memo1.Lines.Add( TSingleton6<TFoo>.GetInstance.DoSomethingImportant);
 Memo1.Lines.Add( TSingleton6<TFoo>.GetInstance.DoSomethingImportant);

 Memo1.Lines.Add( TSingleton6<TBar>.GetInstance.DoSomethingImportant);
 Memo1.Lines.Add( TSingleton6<TBar>.GetInstance.DoSomethingImportant);
end;
```

Conclusão

Muita coisa foi mostrada neste artigo que talvez os programadores acostumados com Delphi 7 nunca tenham visto: class vars, construtores e destrutores de classe, generics. Além disso, foi apresentada a verdade por trás dos construtores: o NewInstance.

Lembre-se que o nome do construtor não precisa ser Create, pode ser qualquer nome, como "Novo", mas ele será um construtor se usar a palavra reservada constructor na frente. O mesmo vale para o Destructor, pode existir um destructor chamado "Mata". Sendo constructor ele usará internamente NewInstance, sem seu conhecimento ou intervenção, e sendo Destructor ele usará da mesma forma o FreeInstance.

O código fonte que acompanha esse artigo contém a implementação completa dos singletons 1 e 2, que são os mais comuns, bem como os outros mais elaborados. Contém também um exemplo em um formulário que testa cada um deles.

Se desejar implementar singletons que simulem herança e que tenham contagem de referências tente usar para isso interfaces, pois estas já vêm com contagem de referências. Isso ficará para um próximo artigo. Como "lição de casa" o leitor pode tentar desenvolver um singleton totalmente "herdável" que em vez de guardar a instância em uma variável ele a guarde numa "hashlist" com referência ao nome da classe e o objeto. Um TStringList pode ser usado como hashlist.

Vitor Luiz Rubio
vitorrubio@gmail.com - http://blog.vitorrubio.com.br
Tecnólogo em Processamento de Dados pela FATEC-SP e MCP. Atua como Analista de Sistemas Sênior na Editora Revista dos Tribunais. Programa com Delphi desde a versão 3 e com .Net desde a versão 1.1.

Coletânea de artigos e links sobre singletons

http://blog.vitorrubio.com.br/2010/11/existem-1001-maneiras-de-preparar.html

http://blog.vitorrubio.com.br/2011/01/existem-1001-maneiras-de-preparar.html

http://blog.vitorrubio.com.br/2011/02/existem-1001-maneiras-de-preparar.html

http://blog.vitorrubio.com.br/2011/02/existem-1001-maneiras-de-preparar_08.html

Discussão sobre necessidade ou não de singletons

http://blog.caelum.com.br/singletons-e-static-perigo-a-vista/

http://forum.springsource.org/showthread.php?12130-Refactoring-from-Singleton-Anti-Pattern-to-Dep-Injection

http://www.martinfowler.com/articles/injection.html

http://guj.com.br/java/38669-singleton-x-metodos-estatiscos

http://forum.springsource.org/showthread.php?12130-Refactoring-from-Singleton-Anti-Pattern-to-Dep-Injection

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Seu **conhecimento vale muito** na DevMedia

Publique artigos e vídeos gratuitamente no site e participe do programa **PayContent.**

Sua participação pode ser premiada: quanto mais você postar, mais chances de ganhar.

Para saber mais acesse:
www.devmedia.com.br/paycontent

DevMedia

Hospede seus sites de forma simples, rápida e ilimitada.

Plano **Básico**

- 1 domínio principal
- 50 domínios hospedados
- Transferência **ilimitada**
- Espaço **ilimitado**
- 10 contas de e-mail
- PHP, ASP e ASP.NET
- Painel de controle Plesk
- Suporte 24hs

Tudo isso por apenas

R$ **15,90** por mês
pagamento trimestral

Assine agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

- Conheça também nossos planos profissionais com registro de domínio grátis e isenção de taxas anuais enquanto for cliente.

HOSPEDIX
HOSPEDAGEM PROFISSIONAL
www.hospedix.com.br

30 DIAS GRÁTIS

Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe 30 dias inteiramente grátis.

RVSTMEDIA